

**ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ**

ДИПЛОМНА РАБОТА

по професия код 481020 „Системен програмист“

специалност код 4810201 „Системно програмиране“

Тема: Уеб приложение за онлайн обяви за покупко-продажба на снимачно и аудио оборудване и предложения за места за снимане на съдържание

Дипломант:
Калоян Стефанов Дойчинов

Дипломен ръководител:
Ангел Пенчев

СОФИЯ

2024



ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

Дата на заданието: 15.11.2023 г. Утвърждавам:..............................
Дата на предаване: 15.02.2024 г. /проф. д-р инж. П. Якимов/

ЗАДАНИЕ
за дипломна работа

ДЪРЖАВЕН ИЗПИТ ЗА ПРИДОБИВАНЕ НА ТРЕТА СТЕПЕН НА ПРОФЕСИОНАЛНА КВАЛИФИКАЦИЯ
по професия код 481020 „Системен програмист“
специалност код 4810201 „Системно програмиране“

на ученика Калоян Стефанов Дойчинов от 12А клас

1. Тема: Уеб приложение за онлайн обяви за попупко-продажба на снимачно и аудио оборудване и предложения за места за снимане на съдържание

2. Изисквания:
   - 2.1. Да се разработи платформа с уеб приложение и със сървърен софтуер
   - 2.2. Платформата да разполага с удостоверяване чрез имейл и парола
   - 2.3. Платформата да разполага с авторизация и роли с различни нива на права
   - 2.4. Потребителите да могат да публикуват обяви за продажба на снимачна и аудио техника
   - 2.5. Потребителите да могат да препоръчват места за снимане на места за създаване на съдържание с интерактивна карта
   - 2.6. Потребителите да могат да качват ревюта и оценки на постове и други потребители
   - 2.7. Платформата да разполага с търсачка на постове и потребители

3. Съдържание
   - 3.1 Теоретична част
   - 3.2 Практическа част
   - 3.3 Приложение

Дипломант :..
/ Калоян Дойчинов /

Ръководител:..
/ Ангел Пенчев /

ВРИД Директор:..
/ ст. пр. д-р Веселка Христова /

Съдържание

Увод

В днешно време създаването на съдържание в интернет става все по-популярно и се превръща в ежедневие за много хора. С издигането на социалните мрежи за предимно аудио и видео споделяне като YouTube, Spotify, Instagram и TikTok, все повече хора създават видеа и подкасти, а те се нуждаят от техника и места за записване.

Качеството на една продукция е от изключително значение за достигането до търсената аудитория. Една част от качеството се изгражда именно от техниката, използвана при създаването на съдържание за социалните мрежи. Тя играе ключова роля във възприемането на крайния резултат.

Също така, не трябва да се подценява и важността на местата, на които се снима. Локациите играят решаваща роля във визуалното излъчване на създаденото. Независимо дали става въпрос за студио с професионално осветление, уникални интериори или живописни пейзажи на открито, правилният избор на място допринася за естетиката и атмосферата на качественото съдържание.

Този дипломен проект - "LensLend", представлява адаптивно динамично уеб приложение, насочено към хора, които желаят да създават дигитално съдържание за интернет. Платформата предоставя онлайн обяви за търсене и предлагане на снимачно и аудио оборудване, както и за предлагане на локации за снимане на съдържание. Целта на "LensLend" е да улесни процеса на създаване на съдържание, предоставяйки достъпна от всяко устройство уеб платформа за намиране на необходимата техника и подходящи места за снимане на снимки, видеа и подкасти.

Глава I

Подходи за създаване на платформата и проучване на подобни системи

1.1. Предимства на уеб приложенията

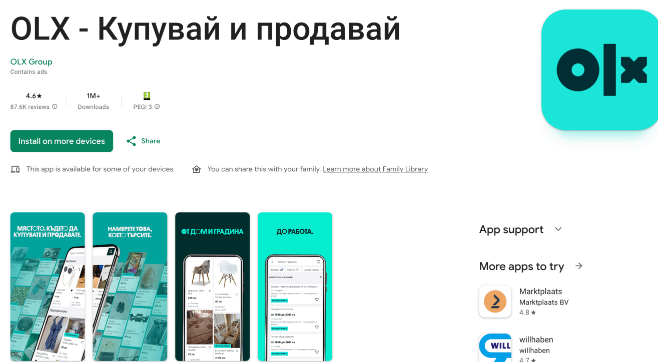
Уеб приложенията предлагат редица ключови предимства, които ги правят популярен избор за разработчици и потребители на онлайн софтуер. Предимствата за разработка на уеб приложение, за разлика от създаване на едноплатформено или многоплатформено мобилно приложение, е, че продуктът е достъпен на всякакви устройства, тъй като има ниво на абстракция – уеб браузър. Всяко устройство с достъп до интернет и уеб браузър може да достъпи уеб приложението без нуждата от инсталиране на допълнителен софтуер. Уеб приложенията също се актуализират централизирано, което означава, че всички потребители винаги използват най-новата версия без нужда от ръчно обновяване. Те са скалируеми и могат лесно да се адаптират към растящия брой потребители и променящите се бизнес изисквания. Разработката и поддръжката на уеб приложения често изисква по-малко време и ресурси в сравнение с традиционния модел на поддръжка на 2 мобилни приложения (Android и iOS) и уебсайт.

Като платформа за покупко-продажба, която постоянно се нуждае от връзка с интернет и минимално съхранение на локални данни, предимствата, които мобилните приложения предлагат, като допълнителна производителност и офлайн достъп до съдържание, не са от първа необходимост.

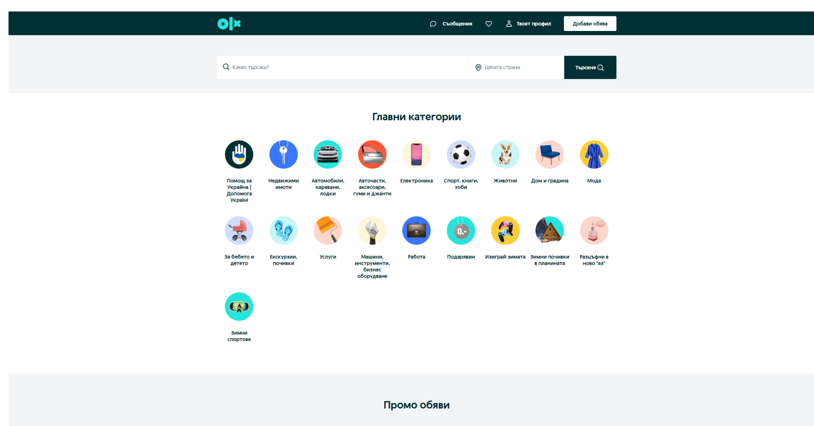
1.2. Съществуващи решения и реализации на пазара

1.2.1. OLX[i] – платформа за онлайн покупко-продажба между потребители

OLX е най-популярната онлайн платформа за онлайн покупко-продажба на стоки втора ръка от всякакъв вид в България и на Балканите. Тя разполага както с уеб приложение *(фиг. 1.1.)*, така и с мобилни приложения *(фиг. 1.2.)* за двете най-големи платформи - Android и iOS. Платформата е локализирана както на български, така и на английски език. Разполага с прост, модерен и чист потребителски интерфейс, в който могат да се публикуват обяви в набор от категории. OLX също дава възможност на потребителите да запазват любимите си обяви, да търсят обяви и да общуват с останалите потребители в платформата чрез тяхна чат система.



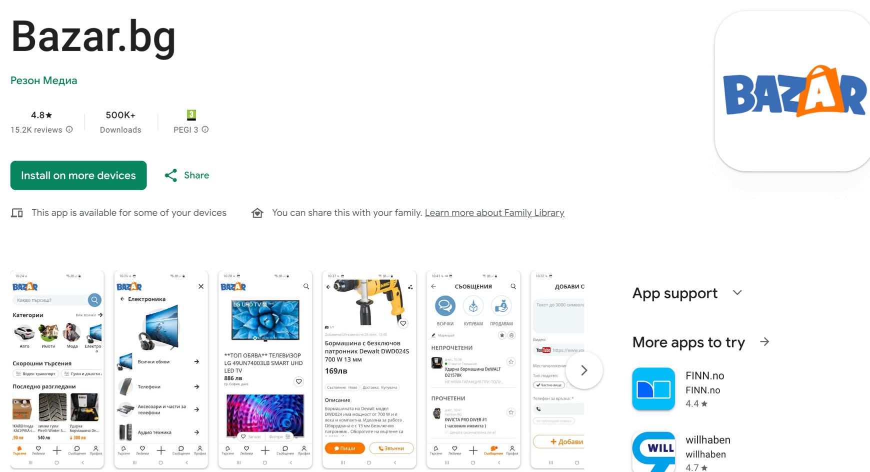
фиг. 1.1. - Мобилно приложение на OLX



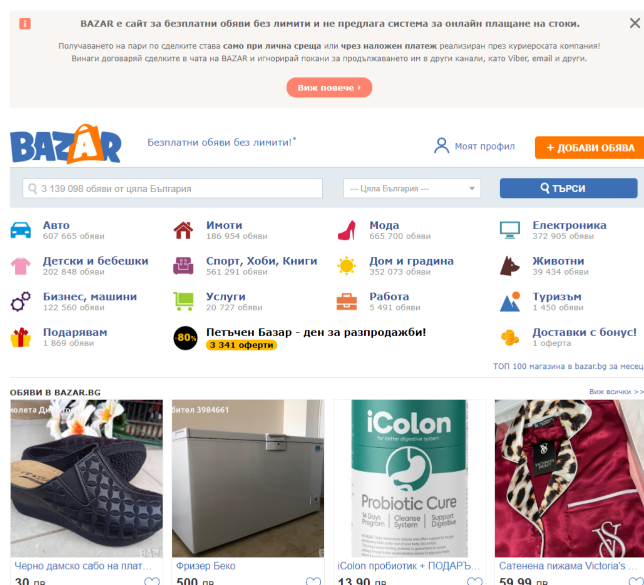
фиг. 1.2. - Уебсайт OLX.bg

1.2.2. Базар.БГ[ii] – платформа за онлайн покупко-продажба между потребители

Базар.БГ е българска разработка и конкуренция на платформата OLX. За разлика от OLX, тя разполага с неограничен (безплатен) достъп до публикуване на обяви. Въпреки че е със сравнително остарял интерфейс[iii], тя продължава да е второто най-посещавано място от българи при търсене и продаване на стоки втора ръка онлайн, тъй като предоставя безплатни услуги на по-централизирано място от обикновен форум.

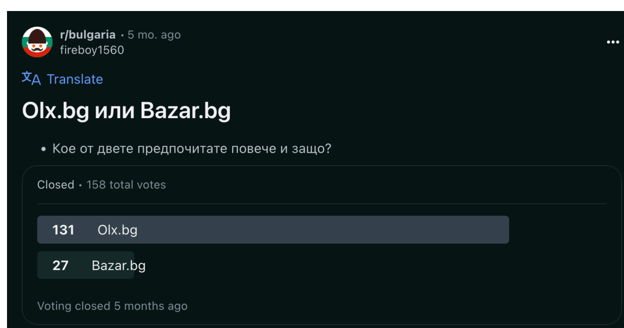


фиг. 1.3. – Мобилно приложение Базар.БГ



фиг. 1.4. - Уебсайт Базар.БГ

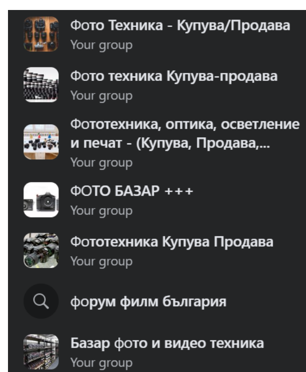
Въпреки това, сред социалните мрежи може да се забележи как OLX има монопол над съзнанието на потребителите на пазара за онлайн продажби на употребявани стоки, бивайки предпочитан от над 82% от българските интернет потребители[iv].



фиг. 1.5. – примерно проучване с гласуване в онлайн платформата Reddit за предпочитание между OLX.bg или Базар.БГ

1.2.3. Facebook групи за препоръчване на места за снимане и продаване и предлагане под наем на оборудване

Често срещано решение на много фотографски проблеми са Facebook групите *(фиг. 1.6.)*, които от една страна ги решават, като събират хората на едно място, но от друга страна ги затрудняват изключително много, ако не са техни членове или не са регистрирани в социалната мрежа, за да използват нейната вътрешна търсачка.



фиг. 1.6. - Топ 7 резултата в платформата Facebook за търсене "фото"

Затова една от целите на дипломната работа е да позволява на търсачките като Google, Bing, Brave, DuckDuckGo и други да събират

данни и т.нар. метаинформация[v], която след това показват на своите потребители, когато търсят даден продукт или място. Това е изключително благоприятно за както начинаещи, така и за хора, които не са толкова запознати с интернет. От друга страна спестява много време търсене на и в частни Facebook групи.

1.3. Методи за разработка на уеб приложение

Уеб приложението е приложение, състоящо се от поне две части - клиент, който обикновено бива уеб браузър или обвивка на такъв, и сървър. Те си комуникират чрез интернет по различни протоколи – HTTP(S)[vi], WebSockets[vii], gRPC[viii] и други.

Има три основни вида архитектури, определени според няколко критерия - потребителско изживяване, скорост на разработване, бързодействие и оптимизации за търсачките – SEO[ix], като те са следните:

1.3.1. Статичен уебсайт[x]

Статичните уебсайтове са популярни и удобни за употреба за новинарски сайтове, блогове, страници на компании и лични портфолиа, където информацията за дадена страница не се променя често. Те са първата технология, използвана за уеб страници, в началото на интернет. В сървър се съхраняват статични файлове като HTML, CSS, снимки и на по-късен етап – JS (JavaScript). Те са изключително леки за всички устройства, включени в уеб протокола – за клиенти, тъй като получават готови страници за показване, за сървърите, тъй като те трябва само да съхраняват файлове и за преносната среда, тъй тези файлове са изключително малки. Тези страници обикновено се зареждат бързо, са

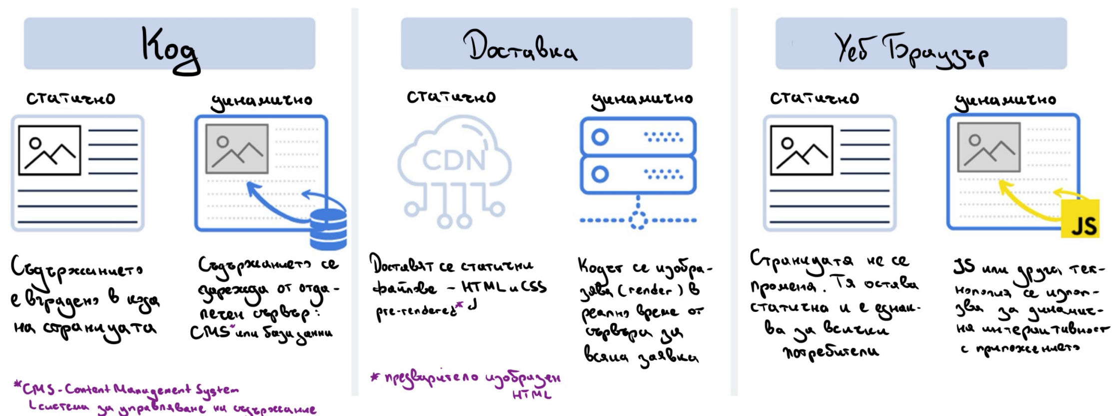
лесни за разработка и поддръжка и не изискват сложна сървърна логика или бази данни.

Въпреки всички тези позитиви, статичните уебсайтове са с ограничена интерактивност, а когато не са, се изисква презареждане на страници или усложняване на първоначалната архитектура както на клиента, така и на сървъра.

В последно време се използват и много модерни технологии, които обикновено се използват за създаване на сървърен уебсайт или уеб приложение с една страница, за създаване и генериране на статични уеб страници и сайтове. Този вид разработка позволява както ускорение на създаването на такива страници (примерно от база данни) чрез автоматизация.

1.3.2. Сървърен (динамичен) уебсайт[xi]

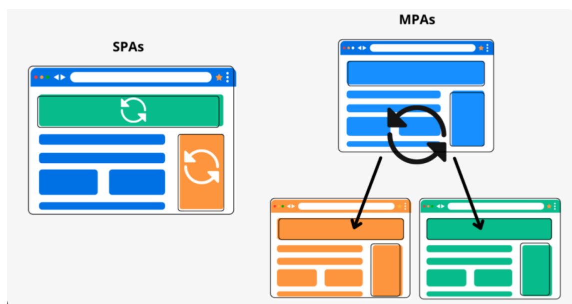
При сървърните уебсайтове (познати още като динамични или сървърно-изобразени[xii]) позволява на уеб страниците да се генерират в реално време в отговор на заявки от потребителя. Това се постига чрез използване на сървърни езици като PHP, Python (с Django, Flask), Ruby (с Ruby on Rails), C# (с ASP.NET) или JavaScript (с Node.js), които работят заедно с база данни за динамично създаване на съдържание и страници.



фиг. 1.7. – Схема на работа на статично и динамично (и техните разлики) изобразяване на уебсайт – свързано с точки 1.3.1. и 1.3.2.

1.3.3. Уеб приложение с една страница[xiii]

Разработването на приложения с една страница (SPA) е метод, при който уеб приложението зарежда една HTML страница и динамично актуализира тази страница при взаимодействие с потребителя. SPA използва софтуерни рамки или библиотеки за JavaScript или TypeScript като Angular, ReactJS или Vue.js за създаване на приложения с богата потребителска интерактивност и производителност, като по този начин се подобрява потребителското изживяване и се увеличава бързината на приложението, тъй като не се изисква презареждане на страницата при промяна на екран, действие или получаване на нови данни.



фиг. 1.8. – Сравнение между едностранично и многостранично приложения

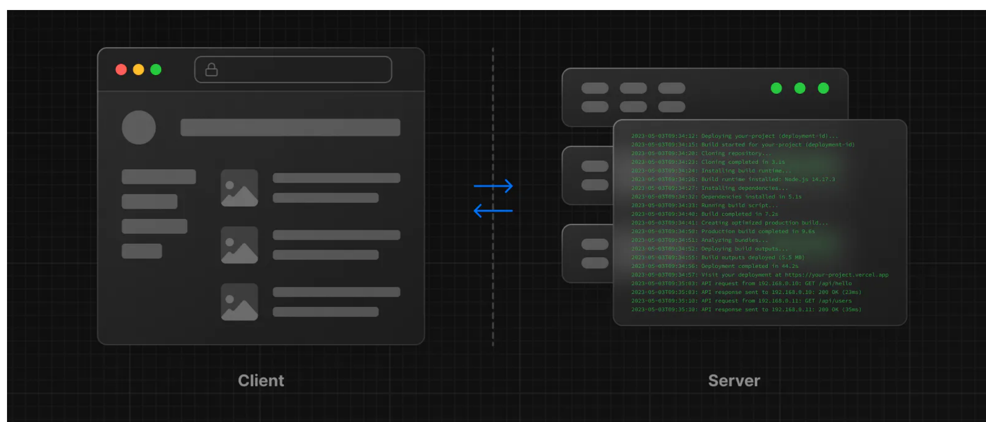
Този тип приложения най-често прави заявки за данни от уеб браузъра на клиента към REST или GraphQL API и вместо да съхранява данни на сървъра, те най-често се съхраняват локално при клиента. Проблеми при приложенията от тип „Една страница“ са, че не са много оптимални за SEO, както и за лимитирани откъм ресурси устройства, поради големия им размер и нуждата всяко приложение да се изобразява и генерира (render) на локалното устройство.

1.3.4. Разработка на модерно хибридно уеб приложение

Хибридната уеб разработка представлява съчетание от сървърно генерирано съдържание (SSR[xiv]) и клиентска интерактивност чрез

клиентски компоненти (CSR[xv]), което позволява на разработчиците да изграждат уеб приложения, оптимизирани както за бързина, така и за потребителско изживяване. Този подход е особено подходящ за създаването на динамични уеб приложения, които се нуждаят от бързо зареждане на първоначалната страница, както и от богати интерактивни функции на клиентската страна.

Популярността на този вид уеб приложения започва с излизането на тестовата функционалност на NextJS 13 през 2022г. за създаване на уеб приложение с компоненти, които се генерират на сървъра (Server Components), и тяхното съчетаване с клиентски такива.

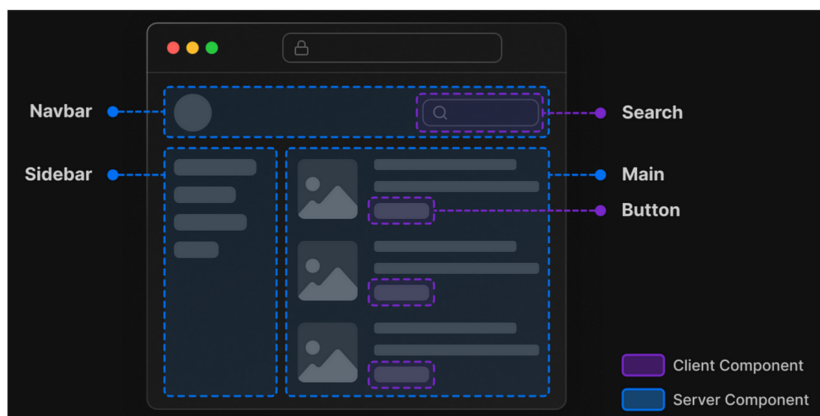


фиг. 1.9. – Изображение показващо двете различни среди на изобразяване на уеб приложение (клиент и сървър)

Сървърните компоненти се използват за динамично генериране на съдържание от сървъра, което позволява бързо първоначално зареждане и подобрена SEO оптимизация, тъй като съдържанието е вече генерирано и изобразено (rendered) при достъп от търсачките. Те често се използват за неинтерактивните компоненти, които обикновено са пълни със съдържание. Клиентските компоненти позволяват богата интерактивност и динамично потребителско изживяване, като се зареждат и изпълняват директно в браузъра на потребителя. Те биват използвани за бутони и компоненти, които зависят от приложно-програмните интерфейси на браузърите.

Тяхната комбинация се използва за създаване на т.нар. хибридни уеб приложения, които обединяват най-доброто от двете среди - бързина и оптимизация от сървърните уебсайтове и интерактивността на уеб приложенията с една страница. Изборът между тях зависи от конкретните нужди на приложението, като целта е да се постигне оптимално балансирано решение.

Въпреки че са сравнително по-сложни за разработка както от динамичен сървърен уебсайт, така и от уеб приложение с една страница, поради нуждата от съображения за предаване на информация между компоненти в различни среди, те полагат новия стандарт за най-оптимални уеб приложения в интерактивна среда.

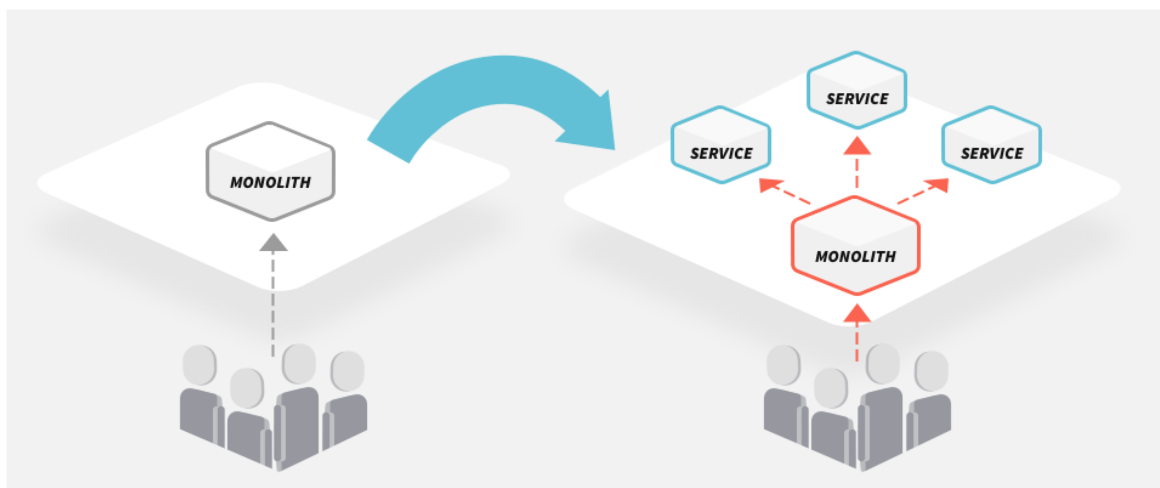


фиг. 1.10. Изображение, показващо примерно хибридно уеб приложение, което използва сървърни (Server – обозначени в синьо) и клиентски (Client – обозначени в лилаво) компоненти.

1.4. Основни архитектури на приложно-програмен интерфейс (ППИ)

Основните архитектури на приложно-програмния интерфейс (ППИ) определят начина, по който се разработва даден сървърен софтуер, неговата комуникация, мащабируемост, ефективност и производителност. В зависимост от специфичните нужди на

приложението, архитектурите на ППИ могат да бъдат класифицирани в две основни категории: монолитна архитектура и архитектура на микроуслугите. Всяка от тези архитектури предлага различни предимства и недостатъци, като изборът между тях зависи от конкретния случай на употреба, изисквания за мащабируемост, гъвкавост, големина и брой на екипи, финансиране и поддръжка.



фиг. 1.11. – Разлика между монолитна архитектура, изобразена с „Monolith”, и архитектура на микроуслугите, изобразена с „Monolith”, който представлява API-Gateway [xvi]*, и “Service”, които представляват услугите*

1.4.1. Монолитна архитектура

Монолитната архитектура за ППИ представлява сървърен софтуер, който е разработен като единно, неделимо цяло. Всички компоненти на приложението, като междинния софтуер, връзката с базата данни, контролерите и цялата бизнес логика, са тясно интегрирани и работят в рамките на един и същ процес / програма.

Предимствата на монолитната архитектура включват лесната разработка в малки екипи, тестване и мащабиране в ранните етапи на разработката на приложението. Освен това, тя предоставя висока степен

на интеграция, бърза комуникация и намалява комплексността, свързана с управлението на множество зависимости (dependencies) и компоненти.

Въпреки това, монолитната архитектура може да представлява предизвикателства при мащабиране или актуализиране на отделни компоненти на приложението. Тъй като всички компоненти са тясно свързани и интегрирани, промяната в един от тях може да наложи необходимостта от тестване на цялото приложение.

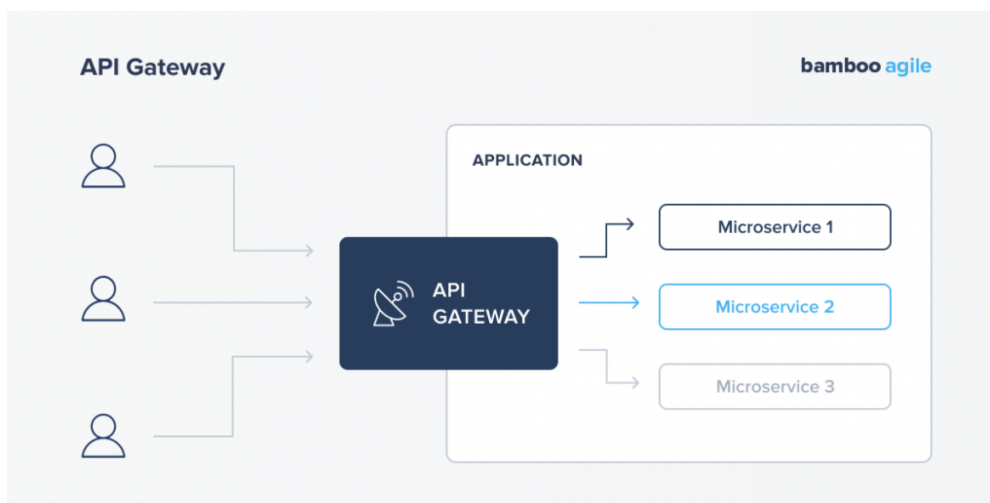
1.4.2. Архитектура на микроуслугите

Архитектурата на микроуслугите е подход за разработка на сървърен софтуер като набор от малки, независими „услуги“, всяка от които работи в свой собствен процес и комуникира с другите миркоуслуги чрез комуникационен протокол като HTTP или специално създаден такъв за такъв тип архитектура - gRPC. Всяка микроуслуга е фокусирана върху изпълнението на точно една конкретна бизнес функция (задача) и може да бъде разработвана, разширявана и мащабирана независимо от останалата част на системата. Което означава, че тя може да бъде оптимизирана дори и на друг програмен език, който е подходящ за дадената функционалност – пр. AI на Python, а останалите функционалности (микроуслуги) на Java Spring.

Това позволява на екипите да разработват и актуализират услуги независимо един от друг, като по този начин се намалява времето за разработка на нови функции. Също така, архитектурата на микроуслугите улеснява мащабирането на отделни части на приложението в отговор на специфични натоварвания, което може да доведе до по-ефективно използване на ресурсите и подобряване на бързината на приложението.

Въпреки тези предимства, архитектурата на микроуслугите представя предизвикателства, свързани с увеличената сложност на

управлението на множество „услуги“, необходимостта от разработване на надеждни механизми за комуникация между тях и гарантиране на сигурност и консистентност в една децентрализирана среда.



фиг. 1.12. – Пример за архитектура на микроуслугите

1.5. Технологии за разработване на уеб приложение

1.5.1. HTML[xvii]

HyperText Markup Language – HTML, е стандартният маркиращ[xviii] език, използван за създаване на съдържание в световната мрежа. То се структурира чрез използването на тагове за определяне на различни елементи като заглавия, параграфи, списъци, връзки, изображения и други мултимедийни обекти.

HTML е и стандарт, който се поддържа от W3C (World Wide Web Consortium), като текущата версия на езика е HTML 5.2.



фиг. 1.13. – Лого на HTML

1.5.2. CSS[xix]

Cascading Style Sheets – CSS, е стандартният стилизиращ език за описание на външния вид и форматирането на документи, написани на HTML или XML. Чрез използването на CSS може да се определят цветове, шрифтове, разстояния между елементите, структура на страницата, и много други аспекти на външния вид, без да е нужна промяна на структурата на HTML документа. CSS също се използва и за създаване на адаптивни уеб дизайни, спрямо размера на екрана на устройство като мобилни телефони, таблети, лаптопи, настолни компютри и телевизори.



фиг. 1.14.- Лого на CSS

1.5.3. JavaScript[xx]

JavaScript е динамичен език за програмиране от високо ниво, поддържащ обектно-ориентиран и функционален стил на програмиране. JavaScript е динамичен програмен език, широко използван за създаване на интерактивни уеб приложения. Той позволява на разработчиците да внедряват сложни функции на уеб страници, като динамично модифициране на съдържанието, контролиране на мултимедия, анимации, и събиране на информация от потребителите. Той е скриптов език, който не се изпълнява от уеб сървър, а от уеб браузъра на потребителя, освен ако не се използва за разработка на приложно-програмен интерфейс с помощта на програмки рамки като Node.js.



фиг. 1.15. – Лого на JavaScript

1.5.4. TypeScript[xxi]

TypeScript е отворен език за програмиране, разработен от Microsoft, който е поддържа всички функционалности на JavaScript и ги надгражда. Той добавя статични типове, интерфейси и класове към JavaScript, което улеснява разработката на по-големи и сложни приложения, като предоставя по-лесен начин за откриване на грешки по време на компилация. Този език за програмиране също може да бъде използван за различни цели като разработването на потребителски интерфейс или за сървърна част. Както JavaScript, така и TypeScript е поддържан от редица програмни рамки като React, Angular и Vue.js Той се компилира до чист JavaScript, което го прави съвместим с всички интерпретатори на JavaScript и уеб браузъри.

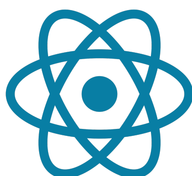


фиг. 1.16. – Лого на TypeScript

1.5.5. ReactJS[xxii]

ReactJS е отворена JavaScript / TypeScript библиотека за създаване на потребителски интерфейси, особено за уеб приложения от тип с една страница. Главната цел на библиотеката е да бъде проста, бърза и мащабируема. React позволява да се създават приложения, които могат да променят своята информация без да има нужда да се презареди страницата. React се фокусира върху ефективността на обновяването на потребителския интерфейс в отговор на промените в данните, като

използва функционален подход и преизползваема компонентна архитектура.



фиг. 1.17. – Лого на ReactJS

1.5.6. Vue.js[xxiii]

Vue.js е прогресивна JavaScript или TypeScript софтуерна рамка за изграждане на потребителски интерфейси. Лесна за интеграция в проекти със съществуващи уеб приложения, тя се фокусира върху декларативното изобразяване и съставянето на компоненти, подобно на ReactJS. Vue.js е проектирана да бъде гъвкава и лесна за използване, като по този начин позволява на разработчиците бързо да създават интерактивни и динамични уеб интерфейси, които се генерират и изобразяват в клиентския браузър.



фиг. 1.18. – Лого на Vue.js

1.5.7. NextJS[xxiv]

Next.js е софтуерна рамка за JavaScript или TypeScript, базирана на ReactJS, която позволява лесно създаване на статични уебсайтове и динамични уеб приложения със сървърно, статично и клиентско генерирано съдържание. Тя предлага оптимизирани предварителни настройки за производителност, снимкова, SEO и мобилна оптимизация,

като същевременно улеснява маршрутизацията, като за разлика от ReactJS има вграден маршрутизатор, оптимизацията на изображенията и интернационализацията.



фиг. 1.19. – Лого на NextJS

1.6. Технологии за разработване на приложно-програмен интерфейс и бази данни

1.6.1. Python[xxv]

Python е мощен, от високо ниво програмен език, който се отличава със своята четимост и яснота в синтаксиса. Широко използван за разработване на уеб приложения, автоматизация, анализ на данни, изкуствен интелект и много други области, Python поддържа множество програмни парадигми, включително обектно-ориентирано, процедурно и функционално програмиране. Благодарение на обширната си стандартна библиотека и активната общност, Python предлага богат набор от инструменти и модули за бързо и ефективно разработване на различни проекти. Python поддържа голямо разнообразие от библиотеки и рамки, разработени от трети страни, като NumPy за научни изчисления, Django и Flask за уеб разработка и TensorFlow за машинно обучение.



фиг. 1.20. – Лого на Python

1.6.2. Django[xxvi]

Django е софтуерна рамка за разработка на сървърни уеб приложения и софтуер за Python, която насърчава бързото разработване и чист, прагматичен дизайн. Предоставя интегрирана система за управление на бази данни, богат набор от библиотеки за обработка на уеб заявки, сесии, удостоверение и авторизация и много други, което я прави предпочитан избор за разработването на комплексни уеб приложения с езика Python.



фиг. 1.21. – Лого на Django

1.6.3. ExpressJS[xxvii]

ExpressJS е бърза, гъвка и минималистична софтуерна рамка за Node.js, предназначена за улесняване на разработката на сървърни уеб приложения и приложно-програмни интерфейси. Тя осигурява минимална и гъвкава структура, която позволява на разработчиците лесно да обработват HTTP заявки и отговори. Една от характеристиките на Express.js е нейната система за маршрутизиране, която позволява да се обработват различни HTTP заявки, въз основа на URL адреса. Тази софтуерна рамка поддържа т.нар. междинни функции, които имат достъп както до заявката, така и до отговора. Тези функции са често използвани при автентикацията или при обработката на грешки. Express е широко използвана заради своята простота и гъвкавост, позволявайки на разработчиците бързо да създават надеждни уеб приложения.



фиг. 1.22. – Лого на ExpressJS

1.6.4. NestJS[xxviii]

NestJS е софтуерна рамка, която надгражда Express, за създаване на ефикасни, мащабируеми Node.js приложения от страна на сървъра. Поддържа напълно TypeScript, като позволява използването на чист JavaScript, комбинира елементи на обектно ориентираното програмиране, функционалното програмиране и функционално реактивното програмиране.

NestJS е вдъхновена както от Angular, така и от Spring. Поддържа инжектиране на зависимости и модули, което улеснява организацията на кода и подобрява мащабируемостта и поддръжката на приложенията.



фиг. 1.23. – Лого на NestJS

1.6.5. MySQL[xxix]

MySQL е популярна система за управление на релационни бази данни (RDBMS), която използва SQL (Structured Query Language) за управление на данните. Тя е с отворен код, като предлага висока производителност, надеждност и гъвкавост, и се използва широко в уеб приложения за съхранение и манипулация на данни. MySQL е предпочитан избор за много разработчици поради своята съвместимост с множество платформи и лесната интеграция с различни програмни езици и технологии.



фиг. 1.24. – Лого на MySQL

1.6.6. PostgreSQL[xxx]

PostgreSQL е мощна, отворена система за управление на обектно-релационни бази данни (RDBMS), която поддържа както SQL (структурирани заявки за език), така и JSON (за нерелационни заявки). Тя е известна със своята напреднала функционалност, надеждност, гъвкавост и поддръжка на големи обеми от данни, което я прави подходяща за сложни приложения, изискващи висока производителност и мащабируемост. PostgreSQL използва MVCC (Version Concurrency Control, за да гарантира, че много потребители могат да имат достъп до едни и същи данни едновременно без конфликти).



фиг. 1.25. – Лого на PostgreSQL

1.7. Интегрирани среди за разработка на уеб приложение и приложно-програмен интерфейс

1.7.1. Visual Studio Code (VS Code)[xxxi]

Visual Studio Code (VS Code) е безплатна, лека и мощна интегрирана развойна среда (IDE) от Microsoft, предназначена за разработка на софтуер. Тя може да се използва на всички популярни операционни системи – Windows, macOS и GNU/Linux. Поддържа множество програмни езици с вградена поддръжка за JavaScript, TypeScript и Node.js, както и разширения за други езици като Python, C++, C#, Rust, Go, Java, PHP и много други. VS Code се отличава с бързото си стартиране, интуитивен интерфейс, вграден Git контрол, и множество разширения, което го прави

предпочитан избор сред разработчиците за различни проекти. Средата осигурява и автоматично довършване на код и предложения, докато пишете, спрямо синтаксиса на езика. Редакторът има вграден терминал, чрез който могат да се изпълняват команди и скриптове директно от него.



фиг. 1.26. – Лого на Visual Studio Code

1.7.2. JetBrains WebStorm[xxxii]

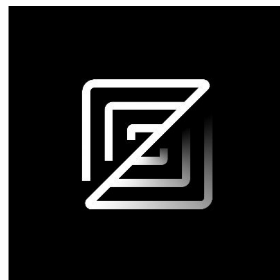
JetBrains WebStorm е мощна интегрирана развойна среда за модерна JavaScript и TypeScript уеб разработка. Поддържа разработка както и на потребителски интерфейс, така и на приложно-програмни интерфейси с Node.js (или негови софтуерни рамки). Предлага много добра поддръжка за ReactJS, NextJS, Angular, Vue.js и други популярни софтуерни рамки за JavaScript и TypeScript. WebStorm осигурява интелигентно писане на код, автоматично довършване на синтаксис, бързо откриване на грешки или предупреждения и изключително лесен рефакторинг[xxxiii]. Тази развойна среда е предпочитана от много разработчици на уеб приложения и сървърен софтуер поради нейната ефективност, много вградени функционалности и удобство при създаването на сложни уеб приложения.



фиг. 1.27. – Лого на WebStorm

1.7.3. Zed[xxxiv]

Zed е високопроизводителен, мултиекранен редактор на код, създаден от създателите на Atom и Tree-sitter. Той комбинира мощта на интегрираните развойни среди с производителността на лек текстови редактор. Zed е проектиран за производителност и ефективно използване на мултиядрени процесори. Тази ИСР излиза на пазара в края на 2022г., като през 2024г. кодът на платформата беше отворен и публикуван. С отварянето на кода на Zed, проектът придоби огромна популярност. Zed е най-модерният редактор на код, който поддържа функции, като автоматично довършване на синтаксис, мулти-буфери, терминал, Vim режим и много други. За жалост Zed е още в началният си етап на разработка, като поддържа само macOS.



фиг. 1.28. – Лого на Zed

Глава II

Функционални изисквания. Избор на софтуерни средства. Проектиране на базата данни

2.1. Функционални изисквания на дипломната работа

- Да се разработи платформа с уеб приложение и сървърен софтуер с NextJS и TailwindCSS за потребителския интерфейс и NestJS за приложно-програмния интерфейс
- Да се разработи база данни с PostgreSQL и времева база данни с Redis
- Платформата да разполага с удостоверение чрез имейл и парола чрез Pasport.js за NestJS
- Платформата да разполага с авторизация и роли с различни нива на права, като модератор и администратор. Където модератор може да променя или изтрива съдържанието на обикновените потребители, а пък администратор да има достъп до защитени администраторски пътища и услуги
- Потребителите да могат да публикуват обяви за продажба или отдаване под наем на снимачна и аудио техника
- Потребителите да могат да препоръчват (създават) места за снимане и места за създаване на съдържание с интерактивна карта на Google Maps
- Потребителите да могат да качват ревюта, оценки или коментари на обяви, места и други потребители
- Платформата да разполага с търсачка на обяви, места и потребители

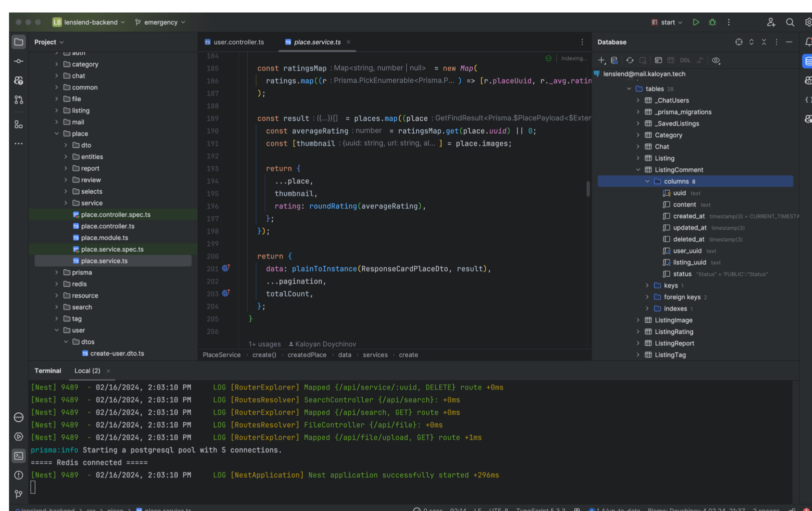
2.2. Избор на софтуерна среда и технологии

2.2.1. JetBrains WebStorm

Интегрираната развойна среда за разработка на софтуер, избрана за изпълнението на тази дипломна работа е, WebStorm, създадена от JetBrains. Нейният избор мотивиран от много добрата поддръжка за JavaScript и TypeScript, както и за техни софтуерни рамки като ReactJS, Angular, Vue.js, Node.js, Express и NestJS.

WebStorm разполага интегрирано откриване на грешки и предупреждение, благодарение на интеграция с ESLint. Други функционалности на средата са: интелигентно автоматично довършаване на код (спрямо синтаксис), навигация и рефакторинг, което значително ускорява процеса на разработка.

Тази среда за разработка е предпочитана заради своята ефективност, удобен интерфейс, много разширения от трети страни и обширни инструменти за разработка, които помагат за повишаване на продуктивността и качеството на кода. Както и дълбока интеграция с други продукти на JetBrains като DataGrip за връзка и визуализиране на бази данни.

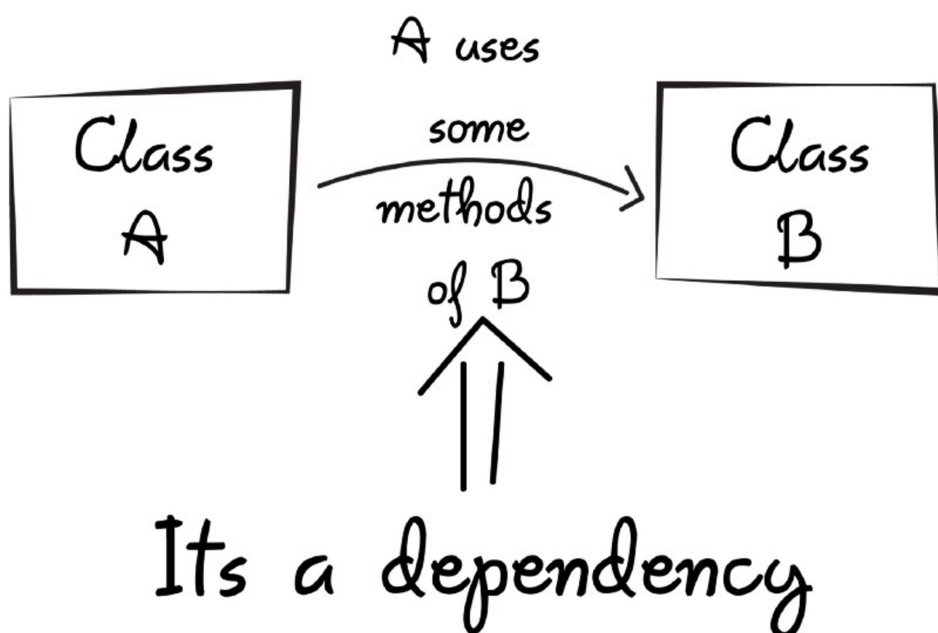


фиг. 2.1. – Отворен проект в WebStorm с отворени множество прозорци за файлова структура, таблици в база данни, терминал и файл.

2.2.2. NestJS

NestJS е избрана за тази дипломна работа за разработка на приложно-програмния интерфейс, поради своите богати и съвременни функционалности и предимства за разработка на сложни, мащабируеми и лесно поддържани сървърни приложения. Като софтуерна рамка, която е базирана на Node.js / Express и използва предимно TypeScript, NestJS предлага ефективна структура за организиране на кода чрез модули, резолвъри и контролери, което улеснява разделението на логиката и повторната употреба на кода.

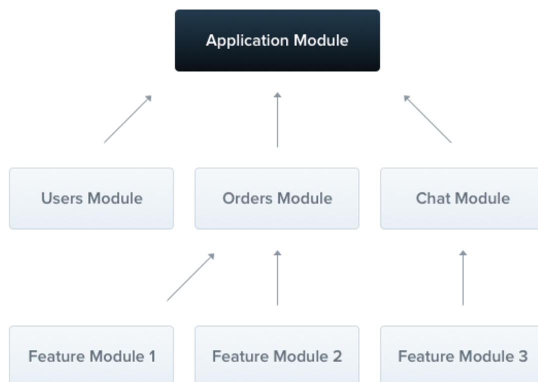
NestJS също поддържа Dependency Injection[xxxv], по подобие на Java Spring който улеснява управлението на зависимостите, а системата за модули спомага с ясната организация на приложението.



фиг. 2.2. – Пример за Dependency Injection

В NestJS за отделните функционалности се създават модули. Всеки модул е изграден от три неща: контролер, service (или още познат като резолвър, в който се намира бизнес логиката) и DTO (Data Access Object), които са модели за пренос на данни между различни процеси и

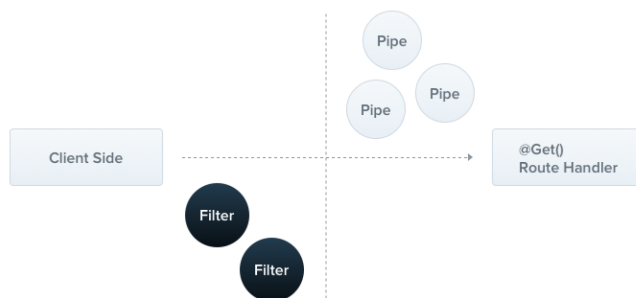
функционалности на даден софтуер. Най-често се използват за информацията, която се предава и получава от външния свят.



фиг. 2.2. – Модели и тяхното разпределение по функционалности

NestJS разполага и с допълнителни функционалности, които изключително много надграждат Express – филтри, pipes, интерсептори и guards.

Филтрите позволяват обработка на грешки (exceptions) за постигане на консистентност в отговорите на грешки, които се връщат на клиенти, както и за осигуряване за постоянно изпълнение на приложението.

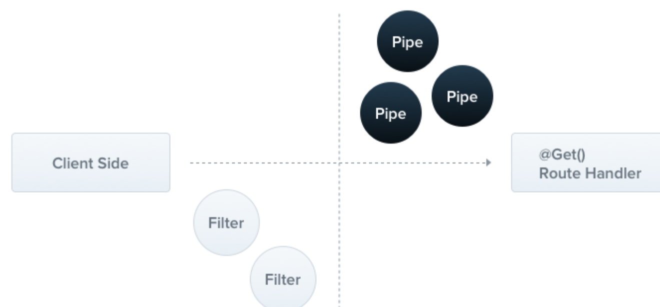


фиг. 2.2. – Филтри

Pipes са класове, които са анотирани[xxxvi] с декоратора *@Injectable()*, който имплементира интерфейса *PipeTransform*. Често, те имат два типични случая на използване:

- За трансформация – преобразуване на входните данни в желаната форма (примерно от стринг към целочислено число)

- За валидиране – сравняват се входните дали са от даден тип, или дали отговарят на дадени изисквания. Ако те са валидни, ще останат непроменени, ако не са, ще се върне грешка на клиента.



фиг. 2.3. - Pipes

Интерсепторите също се анотират с декоратора *@Injectable()*. Те предоставят механизъм за манипулиране на логиката на заявките и отговорите, улеснявайки трансформацията на данни и управлението на операции като кеширане. Те са вдъхновени от концепцията за аспектно ориентирано програмиране, където те дават възможност за:

- Добавяне на логика преди и след изпълнението на дадена функция
- Трансформиране на резултат, вследствие изпълнението на дадена функция
- Трансформиране на грешки вследствие изпълнението на дадена функция
- Допълване и разширяване на логиката на дадена функция
- Или цялостна промяна на логиката на функцията (override), спрямо данните, които са ѝ подадени. Това най-често се използва за кеширане.



фиг. 2.4. – Интерсептори

Guards също се анотират с *@Injectable()*. Те имат само една цел и тя е да осигуряват сигурност, като позволяват лесно управление на разрешенията, правата до достъп и авторизация.



Фиг. 2.5. – Guards

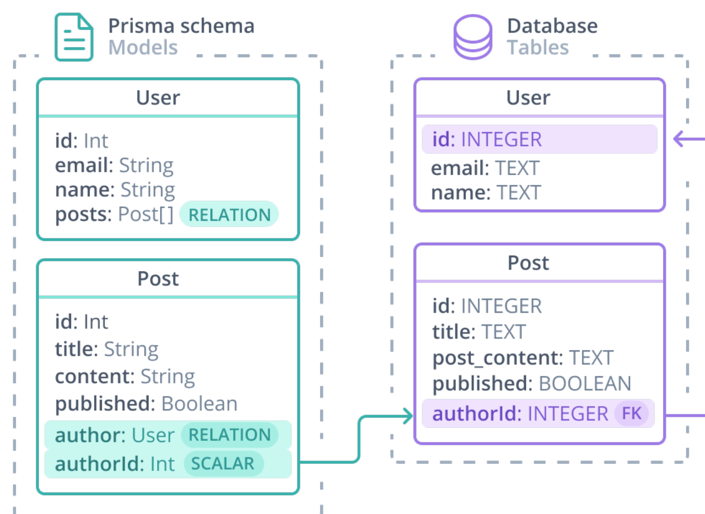
NestJS предлага и вградена поддръжка за създаване на приложно-програмни интерфейси с архитектура на микроуслугите, което позволява лесно създаване на сложни и децентрализирани системи.

2.2.3. Prisma ORM[xxxvii]

ORM[xxxviii] (Object-Relational Mapping) е техника за преобразуване на данни между несъвместими системи в обектно-ориентираното и функционалното програмиране, което позволява на разработчиците да работят с бази данни чрез високо ниво на абстракция – модели, които са типизирани обекти.

Prisma е модерен ORM, който предоставя чудесно изживяване в процеса на разработка (DX)[xxxix], като го прави един от най-предпочитаните ORM софтуери при използване Node.js. Той улеснява работата с бази данни в приложения, разработени с Node.js и неговите софтуерни рамки като NextJS, NestJS и други. Prisma също работи и с много голям набор от най-популярните софтуери за бази данни като MySQL, ProstgreSQL, MariaDB, CockroachDB, MongoDB, SQLite и много други. Той предоставя типизиран, чист и безопасен интерфейс за достъп до базата данни с TypeScript.

Prisma ORM улеснява процеса на моделиране на бази данни, като разработчика трябва да създаде само една схема: „schema.prisma”, в която трябва да опише таблиците, полетата, техните типове и връзки *(фиг. 2.1.)*.



фиг. 2.6. – Prisma модел към база данни

Като след създаване или промяна на тази схема, преди промените да влезнат в ефект, Prisma предоставя изключително опростен интерфейс за създаване на миграции. Това улеснява изключително много създаването на структури на бази данни, тяхното следене и създаване на миграции и други промени. Поради това тя беше избрана за ORM на сървърния софтуер на тази дипломна работа.



фиг. 2.7. – Лого на Prisma

2.2.4. ProstgreSQL

Базата данни, която се използва в тази дипломна работа, е PostgreSQL, поради нейната скорост и оптимизации за съхраняване на множество данни. Тя лесна за използване и има много добре описана и

подробна документация. Има голямата общност, която използва PostgreSQL, която непрекъснато създава допълнителни функции както за самата база данни, така и за инструменти като ORM, с които да се подобрява функционалността и изживяването на разработчика (DX). Освен това PostgreSQL, заедно с MySQL, са с най-голяма поддръжка в Prisma ORM и разполагат с много повече функционалности, включително Full Text Search[xl]. Лично разполагам и с много предишен опит в разработката на бази данни с PostgreSQL.

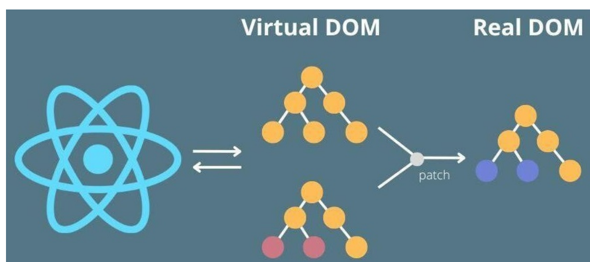
2.2.5. ReactJS

React е JavaScript библиотека, създадена и поддържана от Meta, която се използва за изграждане на потребителски интерфейси. Тя се използва в много от приложенията на Meta като "Instagram for Web", "Meta Account Management" и Facebook. React може да се използва както за създаване на мобилни (чрез React Native), така и на приложения с една страница.

React позволява разделянето и преизползването на код чрез компоненти, които се генерират и визуализират (render) в DOM[xli] на уеб страницата. Компонентите могат да приемат параметри, наречени "props" - свойства, и да бъдат декларирани като функции, връщащи JSX. React използва виртуален DOM за оптимизиране на актуализациите в реалния DOM, визуализирайки (render) само променените части, което увеличава ефективността на приложението, като използва сравнително по-малко ресурси.

JSX е разширение на JavaScript синтаксиса, което улеснява структурирането на интерфейси в React чрез HTML-подобен код. JSX поддържа HTML атрибути и позволява добавянето на персонализирани атрибути, които компонентите получават като свойства (props),

улеснявайки динамичното въвеждане на данни и поведение в уеб приложения.



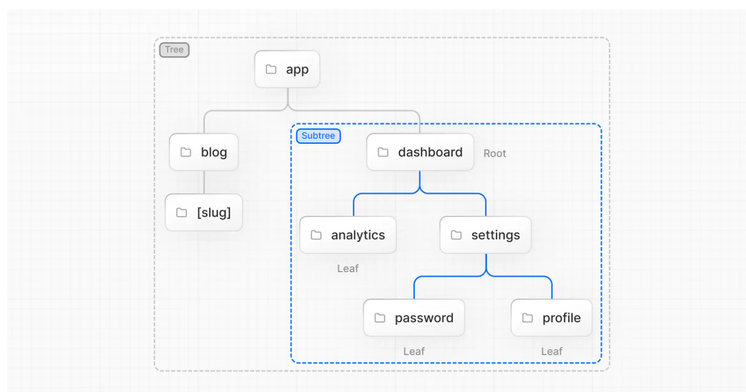
фиг. 2.8. – Как ReactJS използва виртуалния DOM, за да модифицира само променените части

2.2.6. NextJS

За изграждането на потребителската част на дипломния проект е избрана софтуерната рамка на ReactJS за изграждане на цялостни уеб приложения – NextJS, заедно с TypeScript, като език на разработка.

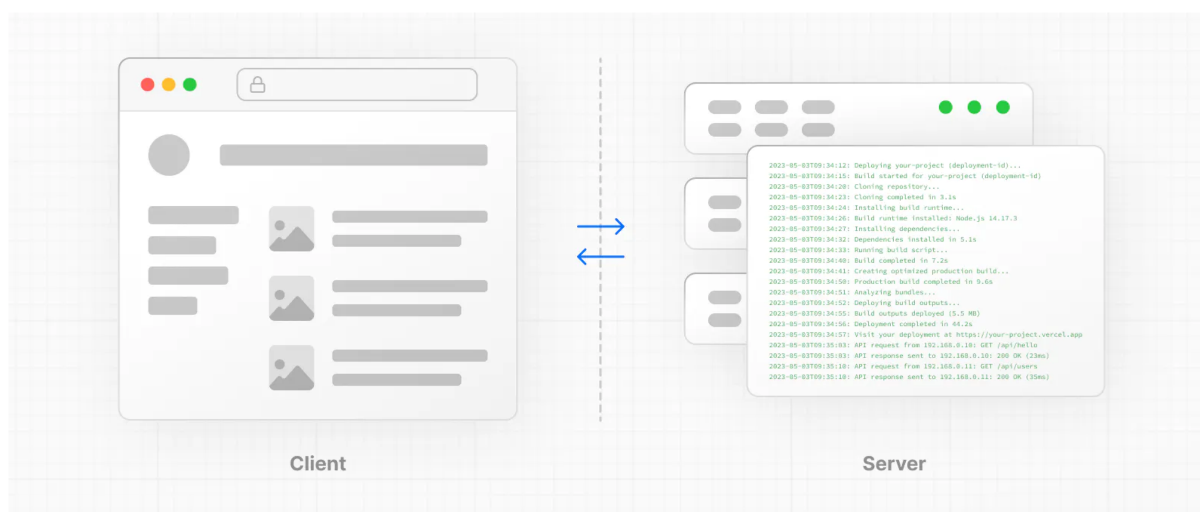
NextJS е разработена от Vercel, тя използва ReactJS компоненти за изграждане на потребителския интерфейс. NextJS абстрахира и автоматично конфигурира инструментите, необходими за React, като пакетиране, компилиране и др. Главните функционалности на NextJS, с които надгражда ReactJS, са:

- Вградена маршрутизация – NextJS, за разлика от ReactJS, където трябва да се използва външна библиотека (като React Router) за маршрутизация на пътищата, използва файлово-базирана маршрутизация.



фиг. 2.9. – Структура на екрани и маршрутизация в NextJS

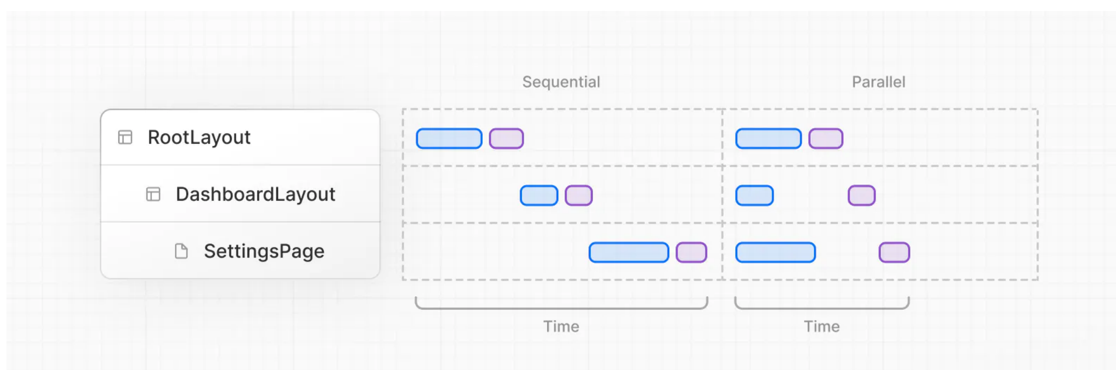
- Смесено генериране и изобразяване (rendering) на страници и компоненти – NextJS по подразбиране използва сървърни React компоненти и използва клиентски или при експлицитно поставяне на „use client" в началото на документа, или при дете на такъв файл. Благодарение на това NextJS се използва за създаването на модерни хибридни уеб приложения, които разполагат както със сървърни и клиентски компоненти, така и със страници, които могат да бъдат изцяло статични, статично или динамично сървърно генерирани и клиентско генерирани. Всичкото това позволява на NextJS да бъде най-оптималната софтуерна рамка за разработване на потребителски интерфейс с JavaScript / TypeScript.



фиг. 2.10. – Клиентско и сървърно генериране и изобразяване (rendering)

- Оптимизация за извличането на данни (fetching) – NextJS надгражда стандартната fetch функция в JavaScript, чрез нейни оптимизации за клиентско и сървърно извличане на данни, като вградено кеширане, меморизация[xlii] и ревалидация[xliii]. Освен това при сървърните компоненти има поддръжка за асинхронно извличане на данни чрез

async / await на JavaScript, както и поддръжка за паралелно сървърно извличане на данни.

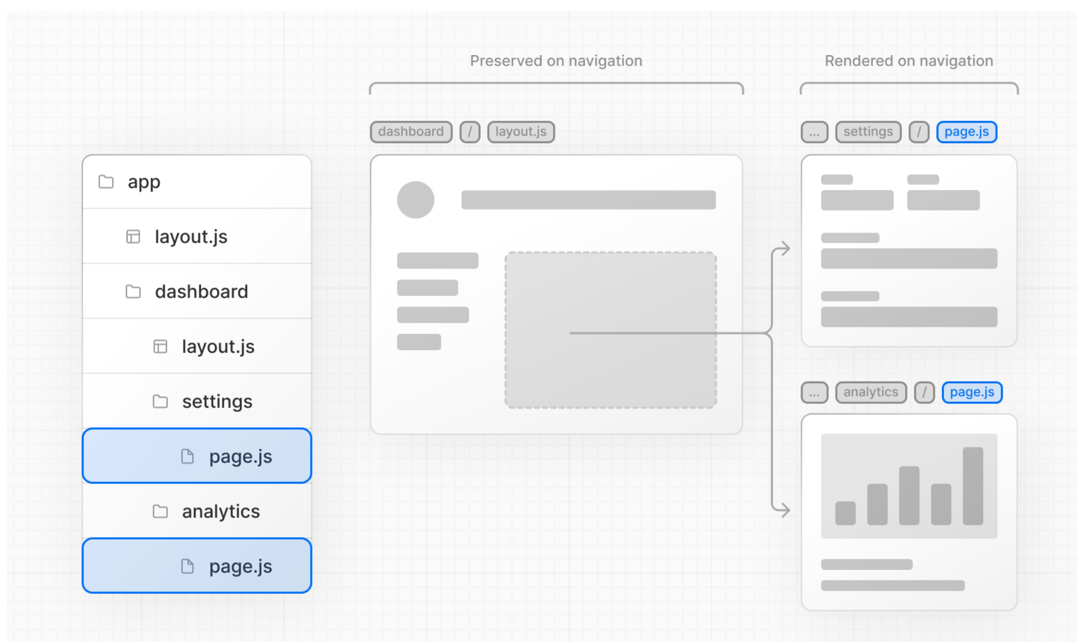


фиг. 2.11. – Разликата между последователно и паралелно извличане на данни на сървъра в NextJS 13+ /app

- NextJS също разполага и с вградена поддръжка за стилизация чрез CSS модули, TailwindCSS, Sass или CSS в JavaScript.

- NextJS разполага с изключително много оптимизации за мултимедийни формати, шрифтове, зареждането и изпълнението на външни скриптове, както и с много интеграции в платформата на създателя на Next – Vercel.

- NextJS взима концепцията на React за преизползване на компоненти и я вдига на друго ниво с layout.js и template.js – специални файлови типове, които се използват за обвивки (консистентни (layout.js) или не (template.js)) около дадени пътища. Те най-често се използват в смисъла на навигация или повтарящ се обвиващ код, или за поставяне на контекст[xliv], който да обгражда всички компоненти под него (показани на *фиг. 2.12.* на следващата страница).



фиг. 2.12. – Структура и шаблонни обвивки в NextJS

- NextJS разполага с още много специални файлове като loading.js (компонент или страница, която се визуализира, докато се извършва извличане на данни), error.js (клиентски компонент-страница, който обвива всички пътища над него, извиква се и се показва на потребителя при грешка, която е настъпила на същото или по-ниско ниво на маршрутизиране в дървото на файловата система, not-found.js, който работи по подобен начин на error.js, но за грешка 404 за ненамерен ресурс. Има и още много други специални файлове, които може да бъдат разгледани в документацията на NextJS.

- Една изключително важна функционалност е, че за всяка страница може да се експортират т.нар. метаданни за страница, които могат да бъдат статични (непроменящи се) или динамични (съдържащи информация от база данни – примерно за страница на даден продукт). Тази функционалност изключително помага както на търсачките за оптимизация SEO, така и за потребителското изживяване, като се споделят различни линкове (страници) в приложения, които

поддържат автоматичен преглед и визуализиране на кратки данни (метаданни) за дадена страница.

2.2.7. Next-intl[xlv]

Next-intl е библиотека за NextJS, създадена отново от Vercel, за i18n (интернационализация) на уеб приложенията. Тя е изключително опростена библиотека, която работи както и за по-нови версии на NextJS (с /app), така и за по-стари (с /pages). Включва и функции за локализация на съобщения, ППИ, базиран на hooks (или сървърни async функции за async сървърни страници) за преводи и форматиране, и инструменти за форматиране на дати, време и числа. Next-intl е проектирана за производителност и е създадена специално за NextJS, по подобие на react-intl, поддържайки международни пътища. Поради това и факта, че Vercel поддържа библиотеката, тя е използвана в тази дипломна работа.

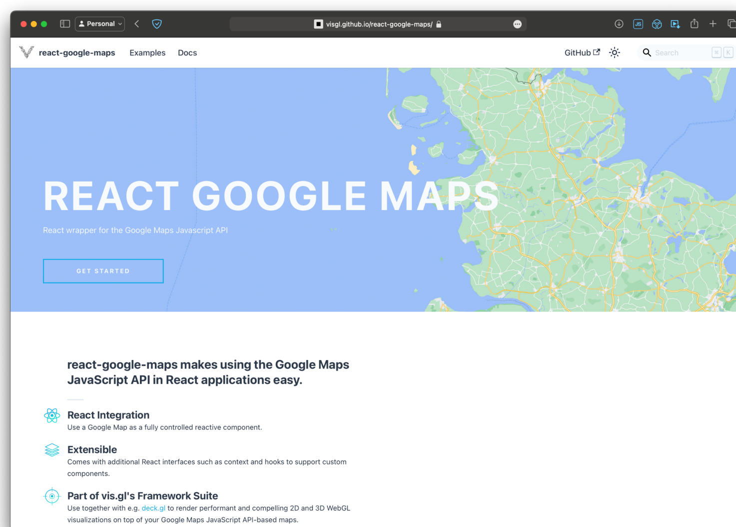


фиг. 2.13. – Лого на next-intl

2.2.8. @vis.gl/react-google-map

@vis.gl/react-google-map[xlvi] е най-модерната, бързата и поддържана нова JavaScript библиотека за Google Maps за използване с ReactJS и TypeScript. Публикувана за първи път на 03.11.2023г. и промотирана от официалните канали на Google Cloud в техния уебсайт, npm.com и популярната платформа за видеосподеляне – YouTube, тя набързо добива популярност ред ReactJS общността. Разработката на vis.gl също сменя

първоначално използваната библиотека в тази дипломна работа – google-map-react[xlvii]



фиг. 2.14. – Уебсайт на библиотеката @vis.gl/react-google-map

2.2.9. SWR[xlviii]

SWR е библиотека, създадена от Vercel, отново, която е създадена и оптимизирана за извличане, кеширане и ревалидиране на данни в клиентски компоненти на ReactJS и NextJS. Името "SWR" произлиза от *stale-while-revalidate* - стратегия за инвалидиране на HTTP кеша, популяризирана от HTTP RFC 5861. SWR е стратегия, при която първо се връщат данни от кеша, когато е възможно. След това се изпращат заявки на определен интервал от време (процес на ревалидиране), за да информацията да бъде актуална. Това е най-популярната библиотека за извличане на данни за клиентски компоненти и е препоръчвана от Vercel като стандарт.



фиг. 2.15. – Лого на SWR

2.2.10. Axios[xlix]

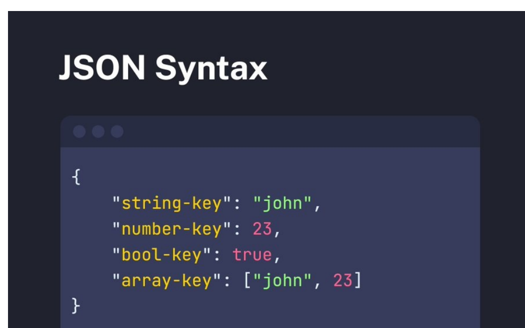
Axios е HTTP клиент, базиран на JavaScript Promises, за NodeJS и браузъра. Използва се извършване на HTTP заявки. Тя предлага лесен за използване ППИ, като улеснява изпращането на асинхронни заявки, като поддържа използването на интерсептори за заявки и отговори, преобразуване на заявки и отговори, и автоматично преобразуване на JSON данни. Библиотеката Axios е избрана заради нейната надеждност и гъвкавост при работа с мрежови заявки. Заедно с SWR на Vercel, те правят една чудесна комбинация за извличане на данни от клиентските компоненти в уеб приложението.



фиг. 2.16. – Лого на Axios

2.2.11. JSON[l]

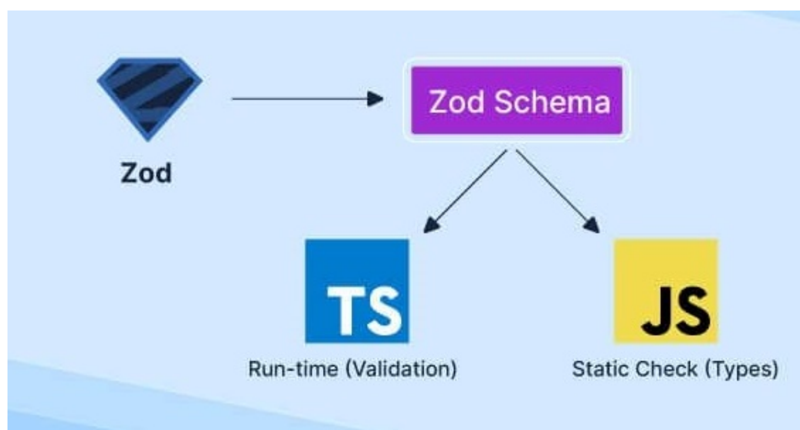
JSON (JavaScript Object Notation) е текстово базиран стандарт за обмен на данни, който е лесен за четене и писане, анализиране и генериране. Форматът на JSON е с лесен синтаксис за представяне на структури от данни и обекти, състоящи се от двойки ключ-стойност. Използва широко в уеб разработката за сериализация и предаване на структурирани данни през мрежата, особено между сървъри и уеб приложения.



фиг. 2.17. – Примерен JSON файл

2.2.12. Zod - [li]

Zod е библиотека за валидация на данни в TypeScript, която улеснява строгото типизиране и валидиране на структури на данни. С Zod, разработчиците могат лесно да дефинират схеми за валидация, които автоматично генерират TypeScript типове, подобрявайки безопасността на типовете и намалявайки вероятността за грешки при работа с данни. Zod се използва в клиентския софтуер за валидиране на данните от потребителския вход преди да бъдат изпратени и валидирани от сървъра.

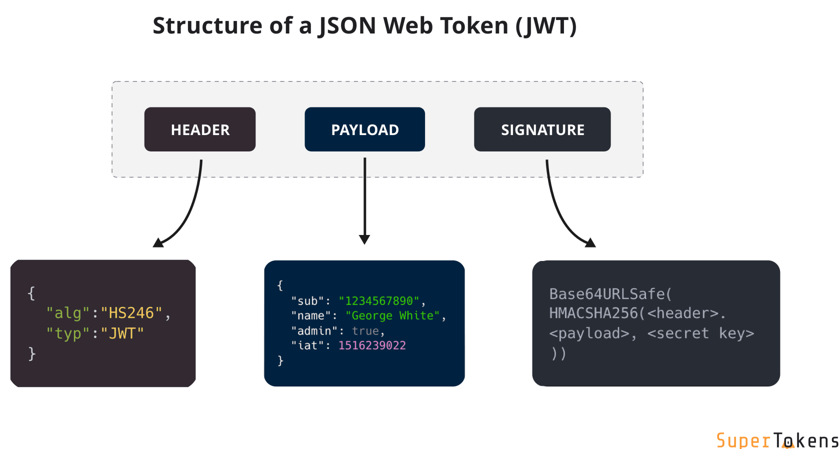


фиг. 2.18. – Как работи Zod

2.2.13. JWT[lii]

JWT (JSON Web Token) е отворен стандарт (RFC 7519), който дефинира компактен начин за сигурно предаване на информация между страни като JSON обект. Информацията може да бъде верифицирана и доверена, тъй като е подписана. Те са широко използвани за удостоверяване и обмен на информация в уеб приложения като токени. Веднъж създаден и върнат на клиента, той може да го използва, за да докаже своята самоличност пред сървъра. Тези токени са подписани от

ключ, намиращ се на сървъра, така че да може да удостовери подписа за валиден.



фиг. 2.19. – Какво се съдържа в примерен JWT

2.2.14. Redis[liii]

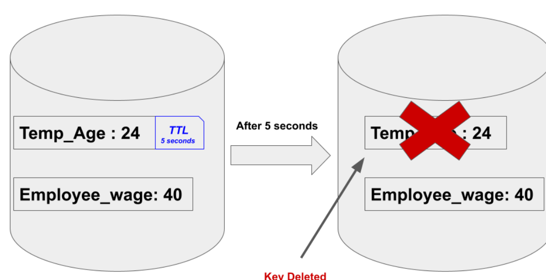
Redis е база данни с отворен код, която се съхранява в оперативната памет, което я прави изключително бърза. Тя се използва за много различни случаи като кеширане на заявки от друга база данни (или ППИ), за стрийимване, за съхранение на документи, като брокер на документи или като векторна база данни. Тя поддържа структури от данни като стрингове, хешове, списъци (листове), обекти, сортирани обекти и поддръжка на сложни заявки.



фиг. 2.20. – Лого на redis

Redis се използва в тази дипломна работа за съхранение на невалидните (забранени) Refresh JWT токени, които ще бъдат по-подробно обяснени в трета глава. Една от причините за нейното използване е, че е изключително бърза и няма по никакъв начин да забави процесите за удостоверение и авторизация на приложно

програмния интерфейс. Но главната причина е, че Redis разполага с една много важна функционалност и тя е TTL – Time To Live, т.нар. време на живот на даден запис, който с всяка секунда се намалява и при достигане на 0 – той се изтрива автоматично. Това е изключително полезна функционалност, тъй като забранените токени изтичат и не бихме искали да съхраняваме изтекли такива, тъй като не само ще заемат място, но и ще забавят и процеса за намиране на такъв запис.



фиг. 2.21. – Примерна функционалност на TTL запис в redis

2.2.15. Amazon AWS S3[liv]

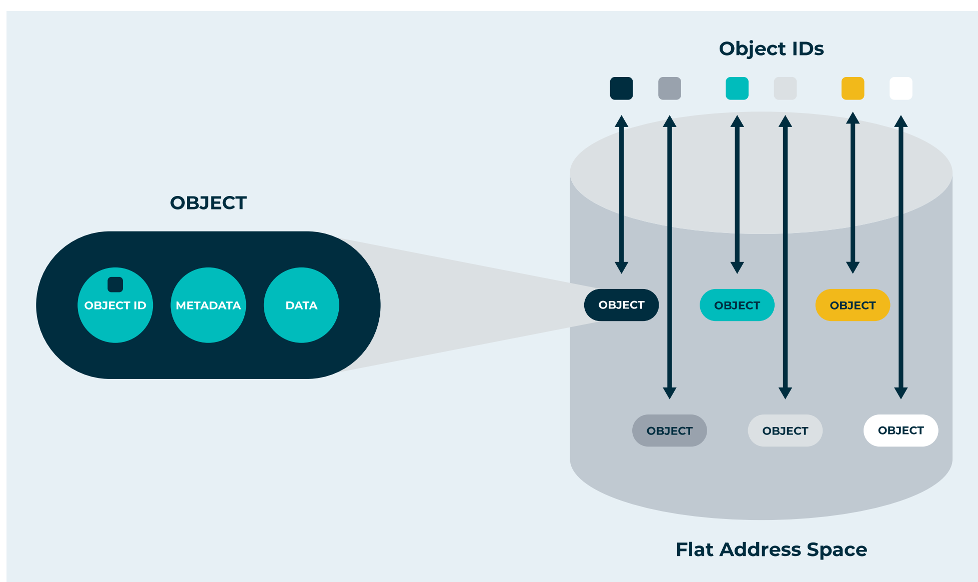
Amazon AWS S3 (Simple Storage Service) е облачна услуга, предлагана от Amazon, която позволява да се съхраняват и извличат файлове (обекти) по начин, който е силно мащабируем и издръжлив. Те могат да бъдат бързо извличани всяка точка на света, благодарение на широкото разпространение на AWS сървърите.



фиг. 2.22. – Лого на Amazon S3

Тази услуга може да бъде използвана за различни нужди, като съхранение на статично мултимедийно уеб съдържание – снимки, смалени снимки (thumbnail - миниатюра), видеа, анимации, иконки и

други. Всички обекти се съхраняват в т.нар. кофи, които са логически контейнери за обектите. Всеки съхраняван обект има уникален ключ (неговия път от корена), който се използва за извличането на обекта.



фиг. 2.23. – Как и какво се съдържа в обектното хранилище и обектите

В тази дипломна работа Amazon S3 се използва заедно с Amazon CloudFront, който представлява CDN[lv] за кеширане и равномерно разпределение на данните в различни сървъри по света.

2.2.16. Google Maps Platform[lvi]

Google Maps Platform предоставя инструменти за множество различни операции с карти. Тази платформа е най-добрата на пазара, поради огромния размер на данни, които Google съхранява и предоставя на своите потребители и разработчици. Функционалности, които се използват в тази дипломна работа са обратното геокодиране и визуализацията на карти, които са интегрирани в най-основните функционалности на уеб приложението – от създаването на места или обяви, в които се избира адрес от интерактивна Google карта, до визуализирането на всички места на една обща Google карта. Обратното

геокодиране се използва за превръщане на координатите на обяви и места в четими адреси на език, избран от потребителя.



фиг. 2.24. – Лого на Google Maps Platform

2.2.17. TailwindCSS[lvii]

TailwindCSS е CSS софтуерна рамка, която позволява стилизирането на уебсайтове и уеб приложения директно в HTML документите чрез класове. Този подход улеснява бързата разработка на собствени дизайни без да е необходимо да се пишат стилове, класове и CSS файлове от нулата, като същевременно се поддържа консистентност и се избягва излишният CSS код. TailwindCSS е предпочитана софтуерна рамка заради своята гъвкавост, ефективност и лесна употреба в модерната уеб разработка. Тя е избрана не само заради тези изброени причини, но и благодарение на дългия ми опит с разработка на различни уеб приложения с TailwindCSS, включително и уебсайта на TUES Fest 2023.

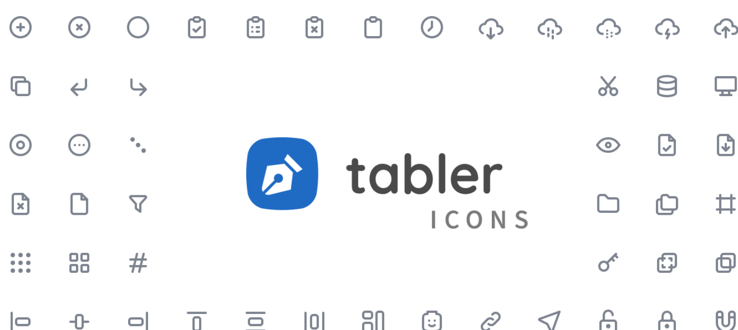


фиг. 2.25. – Лого на TailwindCSS

2.2.18. Tabler Icons[lviii]

Една от най-важните части на едно приложение, независимо дали е уеб или мобилно, е да разполага с консистентен дизайн, което означава,

че трябва да се използват и иконки от една и съща библиотека или колекция. Една от най-богатите такива е Tabler Icons – огромна колекция от безплатни, висококачествени SVG иконки, оптимизирани за уеб дизайн и приложения. Те са проектирани да бъдат прости, изразителни, и най-важното – красиви. Тази колекция е идеална както за разработчици, така и за дизайнери, поради нейната интеграция както в React (и други софтуерни рамки за JS), така и във Figma (под формата на разширение).



фиг. 2.26. – Лого на Tabler Icons

2.2.19. Prettier[lix]

Prettier е инструмент за форматиране на код, който автоматично организира JavaScript, TypeScript, CSS и други поддържани езици, за да направи кода по-четим, консистентен и подреден. Той се интегрира с много редактори и разводни среди за разработка на софтуер, като Visual Studio Code и всички развойни среди на JetBrains. Prettier разполага и с много допълнителни разширения, създадени от трети страни, които спомагат още повече за структурирането на кода. Една от тях е Prettier for TailwindCSS, която сортира и форматира класовете, които съдържат TailwindCSS класове, правейки ги много по четими.



фиг. 2.27. – Лого на Prettier

2.2.20. ESLint[lx]

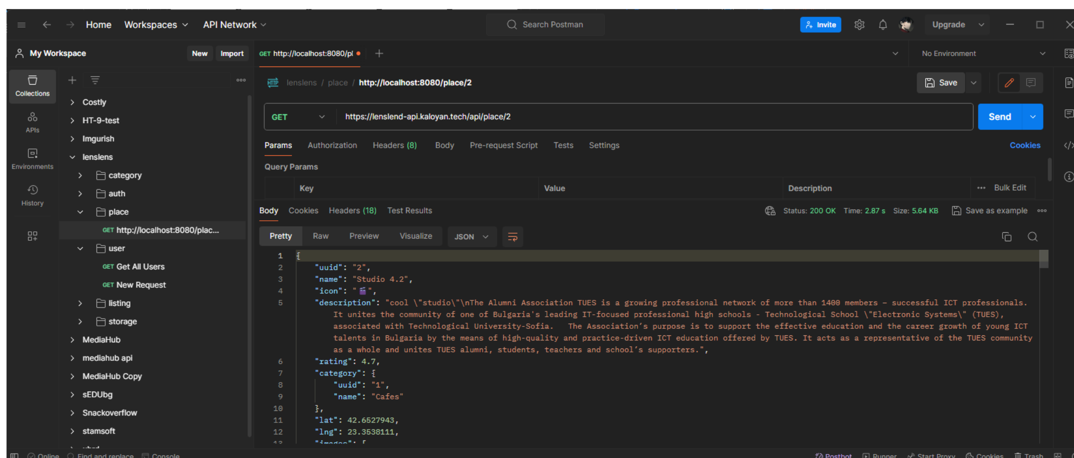
ESLint е инструмент за статичен анализ на код, който помага на разработчиците да откриват и коригират проблеми (грешки и предупреждения) в JavaScript / TypeScript кода на даден, като следват предварително дефинирани правила за стил и качество на кода – които са дефинирани както глобално, така и в локален файл. ESLint разполага с много високо ниво на конфигурация и даже поддържа добавянето на разширения или унаследяването от други проекти или компании – примерно ESLint конфигурацията на Airbnb.



фиг. 2.28. – Лого на ESLint

2.2.21. Postman[lxi]

Postman е най-популярният инструмент за разработване и тестване на приложно-програмни интерфейси. Той улеснява изпращането на заявки към ППИ, визуализацията на отговорите, автоматизацията на тестове и създаването на документация. Също така, Postman предлага функции за работа в екип, което позволява на разработчиците да споделят и управляват заедно тестове и документация с други членове на екипа за тестване и разработване.

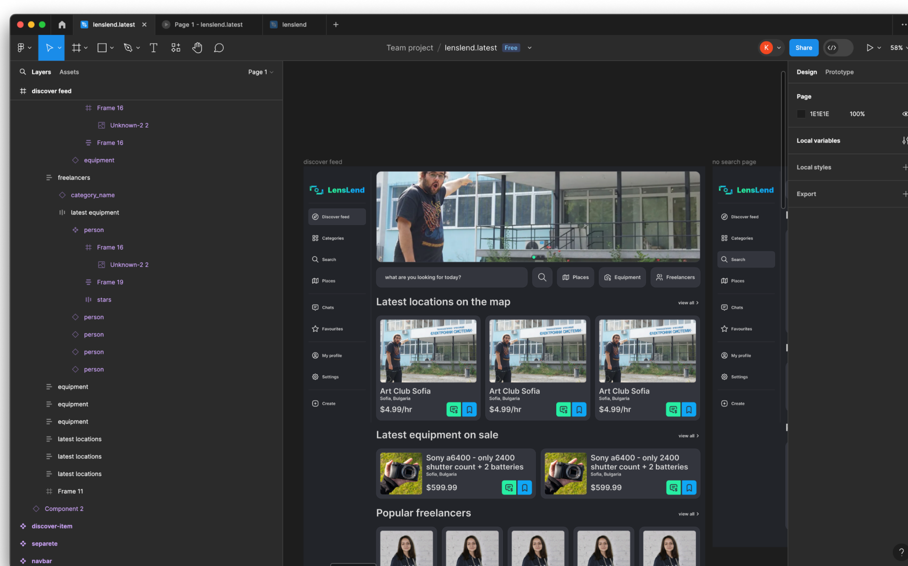


фиг. 2.29. – Примерна заявка в Postman

2.2.22. Figma[lxii]

Figma е най-популярният и прост инструмент за създаване дизайн прототипи, който позволява на дизайнери и екипи да създават, сътрудничат и споделят интерфейси и прототипи в реално време. Той се отличава с интуитивен интерфейс, функции за сътрудничество и вградена поддръжка за дизайн системи, което го прави идеален за създаване на дизайн за уеб и мобилни приложения.

Figma е използвана в този дипломен проект за създаване на ранен дизайн и прототип на интерфейса на уеб приложението (http://tinyurl.com/lenslend-figma). Създаването на дизайн в началото на процеса на разработка на каквото и да е приложение, улеснява следващите етапи на разработка и предоставя консистентност в различните компоненти и страници.



фиг. 2.30. – Екран за разработка във Figma на този дипломен проект

2.2.23. Git[lxiii]

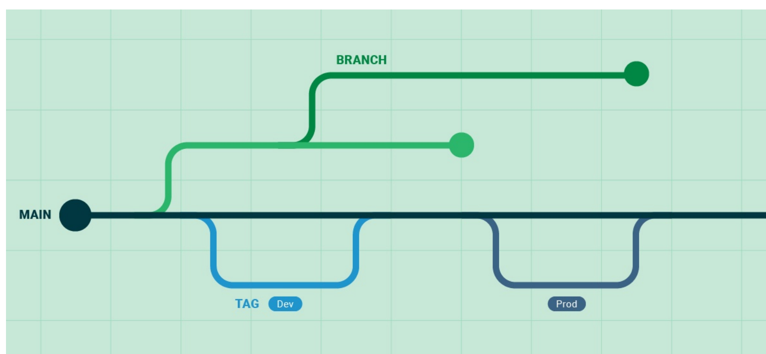
Git е стандартът за система за контрол на версиите[lxiv], която позволява на разработчиците да следят и управляват промените във

файловете на своите проекти. Тя поддържа ефективно сътрудничество между екипите, като предоставя инструменти за сливане на промени (merge), автоматично или ръчно оправяне на конфликти (resolve conflict), връщане към предишни състояния и версии на кода, както и разделение на кода по бранчове за всяка отделна функционалност или логически разделена част от даден проект. Git е една от задължителни системи за използване по време на създаване на дипломна работа, поради нейния голям брой функционалности като следене на промените по кода и особено връщане на промени назад.



фиг. 2.31. – Лого на Git

В разработката на този дипломен проект функционалността за разделение на код – бранчове, е използвана много често в за разделение на различните функционалности на приложно-програмния интерфейс и клиентски софтуер.



фиг. 2.32. – Структура на Git хранилище с бранчове

2.2.24. GitHub[lxv]

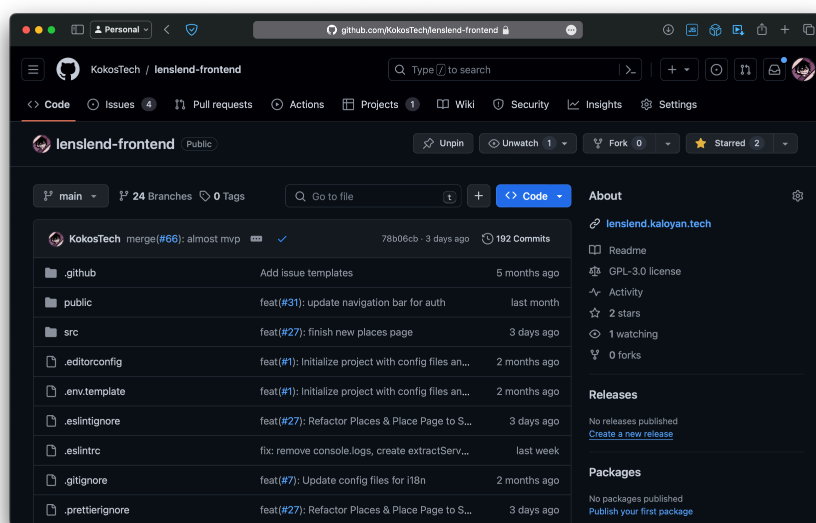
GitHub е уеб-базирана платформа за хостинг на проекти и извадки код, която използва Git за контрол на версиите. Тя позволява съхраняване, споделяне и сътрудничество по проекти и хранилища.

GitHub предлага функции като преглед на кода (code review), управление на проекти (project management), създаването на „проблеми“ (issues), които се използват за заявки за нови функционалности, оправяне на бъгове, документация и т.н.

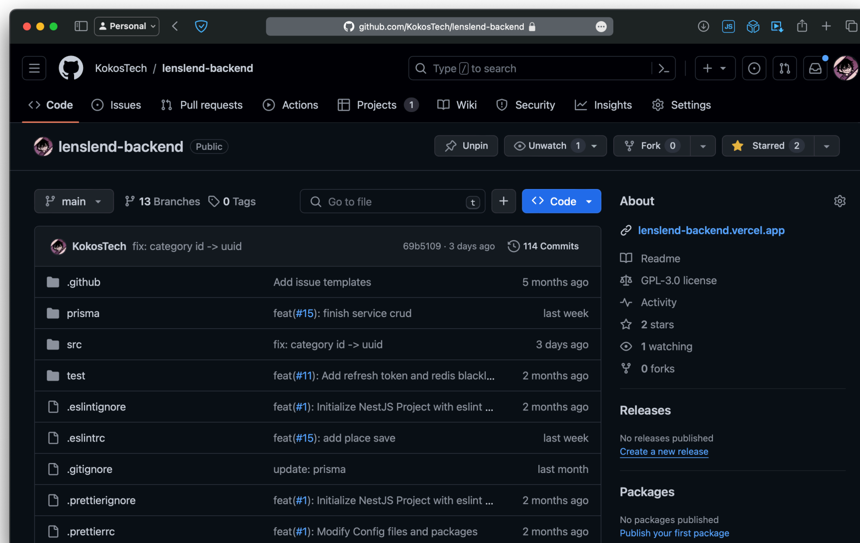


фиг. 2.33. – Лого на GitHub

GitHub позволява и интеграция и автоматизация на проекти чрез GitHub Action, които се използват в тази дипломна работа за проверка на изтекли лични данни (secret) и качване и публикува (deploy) на проекта в облака. В този дипломен проект са използвани 2 GitHub хранилища – за сървърен и клиентски софтуер, които са интегрирани в един проект, в който са описани всички изисквания и задачи (като issues), които трябва да бъдат свършени. За всяка функционалност се създава нов бранч (за даден проблем – issue), като при нейното финализиране се слива (merge) в главния (dev) бранч за разработка чрез git merge и отварянето на pull request.



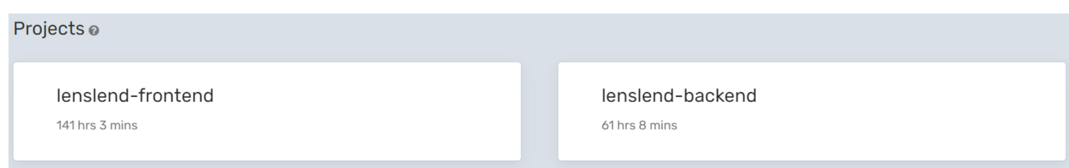
фиг. 2.34. – GitHub хранилище на клиентския софтуер (потребителски интерфейс) на тази дипломна работа - LensLend



фиг. 2.35. – GitHub хранилище на сървърния софтуер (приложно-програмен интерфейс) на тази дипломна работа - LensLend

2.2.25. WakaTime[lxvi]

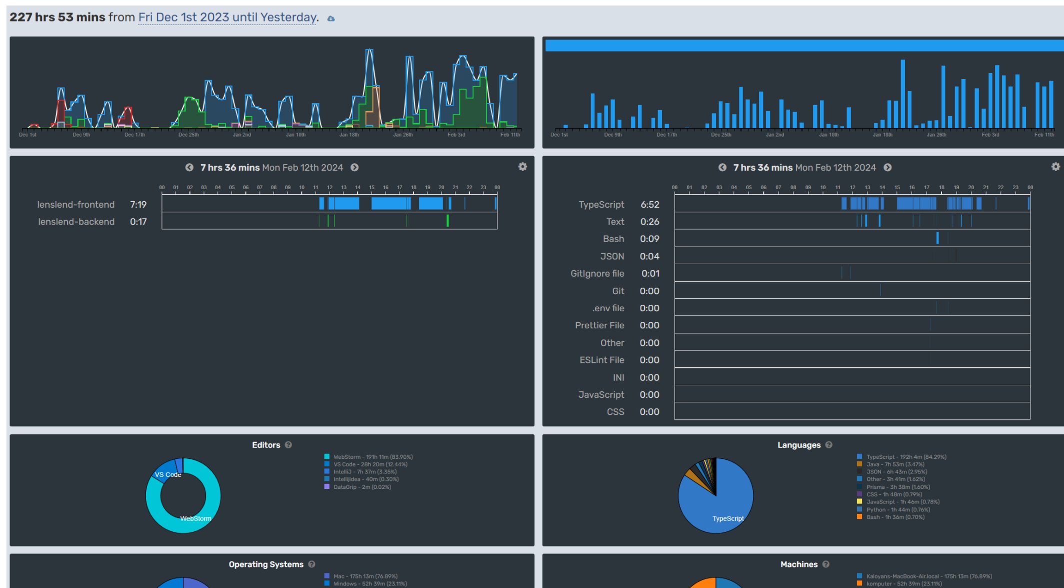
WakaTime е инструмент, използван от програмисти и разработчици, за автоматично проследяване на времето и за мониторинг на тяхната продуктивност и ефективност по време на програмиране и работа на проекти. Тази платформа предоставя подробна статистика за времето, прекарано в различни редактори и проекти, докато се използва клавиатура. WakaTime автоматично отчита времето, прекарано в писане на код, без нуждата от ръчно стартиране или спиране на таймер.



фиг. 2.36. - Време прекарано в писане на проекти, визуализирано и проследено с WakaTime

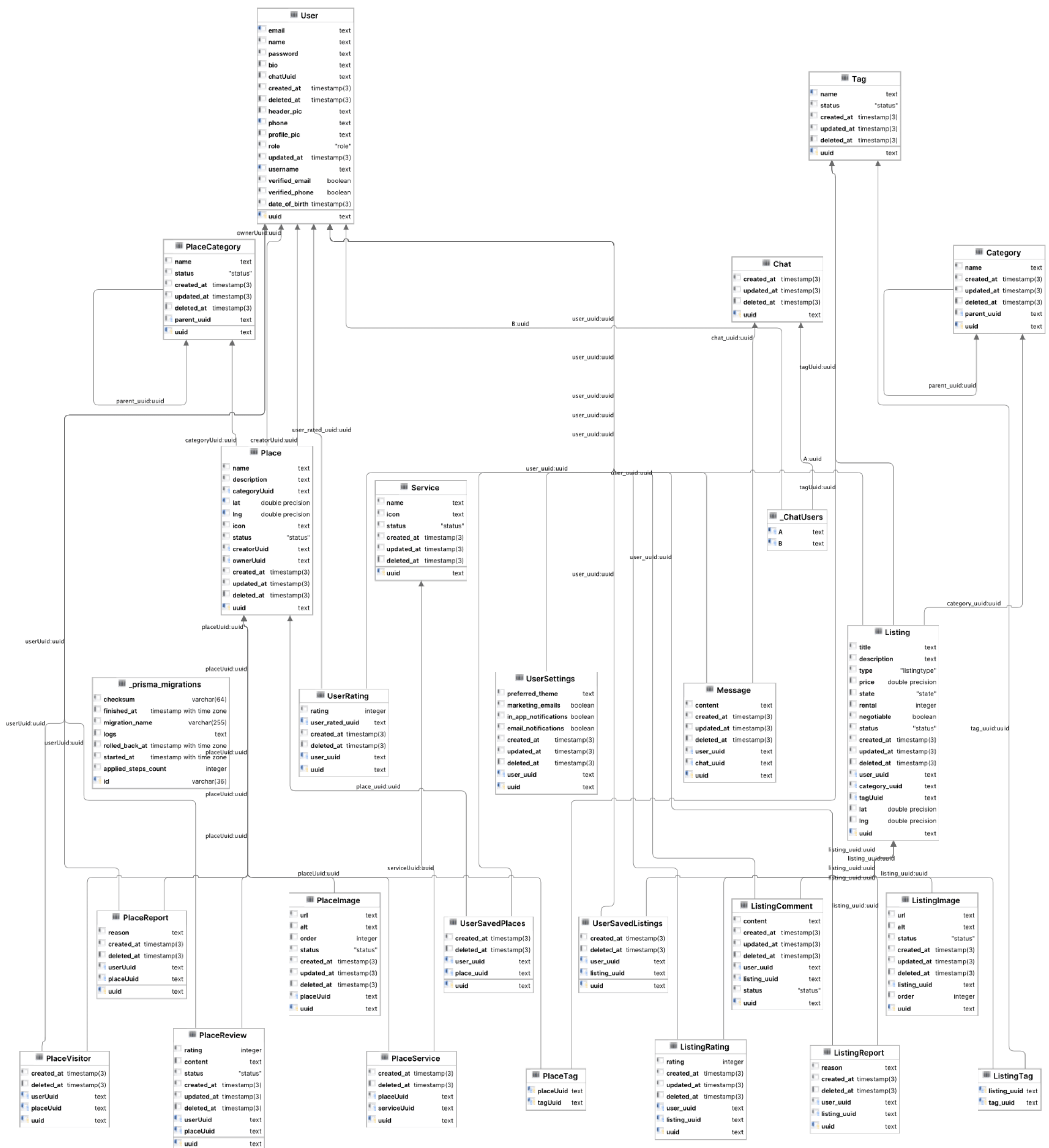
Информацията се анализира и представя в удобни за разбиране графики и отчети, които могат да бъдат използвани за оценка на

продуктивността, планиране на проекти и подобряване на управлението на времето.



фиг. 2.37. - Графично визуализиране на данни за определен период от време: 01.12.2023г. - 12.02.2024г.

2.3. Структура на базата данни



фиг. 2.38. - Схема на база данни

2.3.1. Таблица „User“

Таблица „User” се използва за съхранение на потребителските данни в базата данни. В нея се съдържат идентификатор, име, потребителско име, имейл адрес, телефонен номер, хеширана парола, роля на дадения потребител, неговата профилна снимка, снимка за фон, описание (биография) на профила, дата на раждане, кога е създаден и последно изтриван профила, както и дали са верифицирани имейла и телефона. Към тази таблица, както може да се види в модела на (фиг. 2.Ж.), има много връзки към таблиците, които съдържат цялото съдържание, създадено от потребителя

```
model User {
  uuid              String            @id @default(uuid())
  name              String
  username          String            @unique
  email             String            @unique
  phone             String            @unique
  password          String
  role              Role              @default(USER)
  profile_pic       String?
  header_pic        String?
  bio               String?
  date_of_birth     DateTime          @default(now())
  verified_email    Boolean           @default(false)
  verified_phone    Boolean           @default(false)
  created_at        DateTime          @default(now())
  updated_at        DateTime          @default(now()) @updatedAt
  deleted_at        DateTime?
  settings          UserSettings?
  listings          Listing[]         @relation("UserListings")
  listingComments   ListingComment[]
  listingsRatings   ListingRating[]
  listingReports    ListingReport[]
  chats             Chat[]            @relation("ChatUsers")
  chatUuid          String?
  Message           Message[]
  PlaceCreator      Place[]           @relation("PlaceCreator")
  PlaceOwner        Place[]           @relation("PlaceOwner")
  PlaceVisitors     PlaceVisitor[]    @relation("PlaceVisitors")
  PlaceReview       PlaceReview[]
  PlaceReport       PlaceReport[]
  UserSavedListings UserSavedListings[]
  UserSavedPlaces   UserSavedPlaces[]
  users_rated       UserRating[]      @relation("user_rated")
  users_rated_from  UserRating[]      @relation("user")
}
```

фиг. 2.39. – Модел на таблица „User”

2.3.2. Таблица „Listing“

Таблица „Listing” се използва за съхранение на информацията за всяка обява в базата данни. В нея се съдържат идентификатор, заглавие, описание, местоположение за среща (което се изразява в точни координати), тип на дадена обява, цена (нормална или под наем, или и двете), състояние на продукта, възможност за договаряне на цена, статус на обявата, както и обикновените времеви полета за създаване, промяна и издиране. Таблицата има връзки за създател, категория, снимки, коментари, оценки, сигнали, тагове (много към много) и запазвания от потребители.

```
model Listing {
  uuid             String             @id @default(uuid())
  title            String
  description      String
  lat              Float?
  lng              Float?
  type             ListingType
  price            Float?
  state            State?
  rental           Int?
  negotiable       Boolean            @default(false)
  status           Status             @default(PUBLIC)
  created_at       DateTime           @default(now())
  updated_at       DateTime           @default(now()) @updatedAt
  deleted_at       DateTime?
  user             User               @relation("UserListings", fields: [user_uuid], references: [uuid])
  user_uuid        String
  category         Category           @relation(fields: [category_uuid], references: [uuid])
  category_uuid    String
  images           ListingImage[]
  tags             ListingTag[]
  comments         ListingComment[]
  ratings          ListingRating[]
  reports          ListingReport[]
  Tag              Tag?               @relation(fields: [tagUuid], references: [uuid])
  tagUuid          String?
  UserSavedListings UserSavedListings[]
}
```

фиг. 2.40. - Модел на таблица "Listing"

2.3.3. Таблица „Category“

Таблица „Category” се използва за динамично създаване на категории в уеб приложението от администратор. В таблицата се

съдържат идентификатор, име, нормалните времеви полета, циклична релационна връзка към категория (която се използва за подкатегории) и връзки към обяви.

```
model Category {
  uuid          String     @id @default(uuid())
  name          String
  created_at    DateTime   @default(now())
  updated_at    DateTime   @default(now()) @updatedAt
  deleted_at    DateTime?
  parent        Category?  @relation("SubCategories", fields: [parent_uuid], references: [uuid])
  parent_uuid   String?
  sub_categories Category[] @relation("SubCategories")
  listings      Listing[]
}
```

фиг. 2.41. - Модел на таблица "Category"

2.3.4. Таблица „ListingImage”

Таблицата „ListingImage” се използва за съхранение на линковете, алтернативния текст на снимките и тяхната подредба. Освен това тя съдържа статус и обикновените времеви полета. Има връзка към таблица със снимки, която е много към едно.

```
model ListingImage {
  uuid        String    @id @default(uuid())
  url         String
  alt         String?
  order       Int?
  status      Status    @default(PUBLIC)
  created_at  DateTime  @default(now())
  updated_at  DateTime  @default(now()) @updatedAt
  deleted_at  DateTime?
  listing     Listing   @relation(fields: [listing_uuid], references: [uuid])
  listing_uuid String
}
```

фиг. 2.42. – Модел на таблица „ListingImage“

2.3.5. Таблица “ListingTag”

Таблицата „ListingTag” се използва като междинна таблица за създаване на много към много връзка между таблиците „Listing” и „Tag”. Това обикновено може да бъде постигнато в Prisma ORM, като тя автоматично генерира такава междинна таблица, но целта е за бъдеще да се добавят други полета.

```
model ListingTag {
  listing      Listing @relation(fields: [listing_uuid], references: [uuid])
  listing_uuid String
  tag          Tag     @relation(fields: [tag_uuid], references: [uuid])
  tag_uuid     String

  @@id([listing_uuid, tag_uuid])
}
```

фиг. 2.43. – Модел на таблица „ListingTag”

2.3.6. Таблица “Tag”

Таблица „Tag” се използва за съхранение на тагове, за да бъдат преизползвани, ако вече съществуват, за да се спести място за съхраняване, повтаряне на данни, както и улеснено индексиране. Таговете съдържат идентификатор, име, което може да е само уникално, статус и обикновените времеви полета, както и с много към много връзки с места и обяви.

```
model Tag {
  uuid       String      @id @default(uuid())
  name       String      @unique
  status     Status      @default(PUBLIC)
  created_at DateTime    @default(now())
  updated_at DateTime    @default(now()) @updatedAt
  deleted_at DateTime?
  listings   Listing[]
  ListingTag ListingTag[]
  PlaceTag   PlaceTag[]
}
```

фиг. 2.44. – Модел на таблица „Tag”

2.3.7. Таблица “ListingComment”

Таблицата „ListingComment” се използва за коментарите под обявите. В нея се съдържат идентификатор, съдържание, статус и обикновените времеви полета. Таблицата е с връзки едно към много с таблиците „User” и „Listing”.

```
model ListingComment {
  uuid         String    @id @default(uuid())
  content      String
  status       Status    @default(PUBLIC)
  created_at   DateTime  @default(now())
  updated_at   DateTime  @default(now()) @updatedAt
  deleted_at   DateTime?
  user         User      @relation(fields: [user_uuid], references: [uuid])
  user_uuid    String
  listing      Listing   @relation(fields: [listing_uuid], references: [uuid])
  listing_uuid String
}
```

фиг. 2.45. – Модел на таблица „ListingComment”

2.3.8. Таблица “ListingRating”

Таблицата „ListingRating” се използва за оценяване на обяви / услуги. Тя прилича много на „ListingComment”, но с две значителни разлики – вместо със съдържание, тя разполага с рейтинг. Освен това е създаден уникален индекс, позволявайки на един потребител да поставя само една оценка за един пост.

```
model ListingRating {
  uuid         String    @id @default(uuid())
  rating       Int
  created_at   DateTime  @default(now())
  updated_at   DateTime  @default(now()) @updatedAt
  deleted_at   DateTime?
  user         User      @relation(fields: [user_uuid], references: [uuid])
  user_uuid    String
  listing      Listing   @relation(fields: [listing_uuid], references: [uuid])
  listing_uuid String

  @@unique([user_uuid, listing_uuid])
}
```

фиг. 2.46. – Модел на таблица „ListingRating”

2.3.9. Таблица “Place”

Таблицата „Place” се използва за съхранение на информацията за всяко място в базата данни. В нея се съдържат идентификатор, име, описание, локация, иконка, статус и обикновените времеви полета. Освен това има връзки за създател, собственик (ако е посочен такъв) категория, снимки, услуги (много към много), ревюта, сигнали, тагове (много към много), хора, които са посещавали, и запазвания от потребители

```
model Place {
  uuid           String          @id @default(uuid())
  name           String
  description    String
  categoryUuid   String
  category       PlaceCategory   @relation(fields: [categoryUuid], references: [uuid])
  lat            Float
  lng            Float
  icon           String
  status         Status          @default(PUBLIC)
  creatorUuid    String
  creator        User            @relation(fields: [creatorUuid], references: [uuid], name: "PlaceCreator")
  ownerUuid      String?
  owner          User?           @relation(fields: [ownerUuid], references: [uuid], name: "PlaceOwner")
  created_at     DateTime        @default(now())
  updated_at     DateTime        @default(now()) @updatedAt
  deleted_at     DateTime?
  services       PlaceService[]
  images         PlaceImage[]
  tags           PlaceTag[]
  reviews        PlaceReview[]
  reports        PlaceReport[]
  visitors       PlaceVisitor[]  @relation("PlaceVisitors")
  UserSavedPlaces UserSavedPlaces[]

  @@unique([lat, lng])
}
```

фиг. 2.47. – Модел на таблица „Place”

2.3.10. Таблица “PlaceCategory”

Таблицата „PlaceCategory” е изградена по подобие на „ListingCategory” само че с връзка много към едно към таблица „Place”

вместо „Listing”. Тя притежава същите полета и циклична връзка за подкатегориите.

```
model PlaceCategory {
  uuid           String         @id @default(uuid())
  name           String
  status         Status         @default(PUBLIC)
  created_at     DateTime       @default(now())
  updated_at     DateTime       @default(now()) @updatedAt
  deleted_at     DateTime?
  parent         PlaceCategory?  @relation("SubCategories", fields: [parent_uuid], references: [uuid])
  parent_uuid    String?
  places         Place[]
  sub_categories PlaceCategory[] @relation("SubCategories")
}
```

фиг. 2.48. – Модел на таблица „PlaceCategory”

2.3.11. Таблица “PlaceService”

Таблицата „PlaceService”, по подобие на „ListingTag” се използва като междинна таблица за създаване на много към много връзка между таблиците „Place” и „Service”.

```
model PlaceService {
  uuid        String    @id @default(uuid())
  created_at  DateTime  @default(now())
  deleted_at  DateTime?
  place       Place     @relation(fields: [placeUuid], references: [uuid])
  placeUuid   String
  service     Service   @relation(fields: [serviceUuid], references: [uuid])
  serviceUuid String
}
```

фиг. 2.49. – Модел на таблица „PlaceService”

2.3.12. Таблица “Service”

Таблицата „Service” се използва за динамично създаване на услуги в уеб приложението от администратор. В таблицата се съдържат

идентификатор, име, иконка, статус и нормалните времеви полета. връзки към обяви.

```
model Service {
  uuid         String         @id @default(uuid())
  name         String
  icon         String
  status       Status         @default(PUBLIC)
  created_at   DateTime       @default(now())
  updated_at   DateTime       @default(now()) @updatedAt
  deleted_at   DateTime?
  PlaceService PlaceService[]
}
```

фиг. 2.50. – Модел на таблица „Service”

2.3.13. Таблица “PlaceImage”

Таблицата „PlaceImage” е изградена по подобие на „ListingImage” само че с връзка много към едно към таблица „Place” вместо „Listing”. Тя притежава същите полета.

```
model PlaceImage {
  uuid       String    @id @default(uuid())
  url        String
  alt        String?
  order      Int?
  status     Status    @default(PUBLIC)
  created_at DateTime  @default(now())
  updated_at DateTime  @default(now()) @updatedAt
  deleted_at DateTime?
  place      Place     @relation(fields: [placeUuid], references: [uuid])
  placeUuid  String
}
```

фиг. 2.51. – Модел на таблица „PlaceImage”

2.3.14. Таблица “PlaceTag”

Таблицата „PlaceTag” е изградена по подобие на „ListingTag” само че тя се използва за много към много връзка между таблиците „Place” и „Tag”

вместо таблиците „Listing” и „Tag”. Тя притежава същите полета и уникална индексация, за да не се повтарят таговете в един пост.

```
model PlaceTag {
  place     Place  @relation(fields: [placeUuid], references: [uuid])
  placeUuid String
  tag       Tag    @relation(fields: [tagUuid], references: [uuid])
  tagUuid   String

  @@id([placeUuid, tagUuid])
}
```

фиг. 2.52. – Модел на таблица „PlaceTag”

2.3.15. Таблица “PlaceVisitor”

Таблицата „PlaceVisitor” е помощна таблица, подобна на таблиците за запазване на обяви и места, за изграждане на връзка между потребител и място. Тази връзка се използва, за да потребителя да може да се отбележи, че е бил на дадено място. Тя съдържа идентификатор и времеви полета за създаване и изтриване.

```
model PlaceVisitor {
  uuid       String    @id @default(uuid())
  created_at DateTime  @default(now())
  deleted_at DateTime?
  user       User      @relation("PlaceVisitors", fields: [userUuid], references: [uuid])
  userUuid   String
  place      Place     @relation("PlaceVisitors", fields: [placeUuid], references: [uuid])
  placeUuid  String

  @@unique([placeUuid, userUuid])
}
```

фиг. 2.53. – Модел на таблица „PlaceVisitor”

2.3.16. Таблица “PlaceReview”

Таблицата „PlaceReview” е изградена по подобие на „ListingComment”, обаче с връзка едно към много с таблица „Place” и

връзка към таблицата „User". В таблицата се съдържат идентификатор, оценка, коментар (ревю), което не е задължително, статус и обикновените времеви полета.

```
model PlaceReview {
  uuid          String          @id @default(uuid())
  rating        Int
  content       String?
  status        Status          @default(PUBLIC)
  created_at    DateTime        @default(now())
  updated_at    DateTime        @default(now()) @updatedAt
  deleted_at    DateTime?
  user          User            @relation(fields: [userUuid], references: [uuid])
  userUuid      String
  place         Place           @relation(fields: [placeUuid], references: [uuid])
  placeUuid     String

  @@unique([placeUuid, userUuid])
}
```

фиг. 2.54. – Модел на таблица „PlaceReview"

2.3.17. Таблица "UserSavedListings"

Таблицата „UserSavedListings" се използва като междинна таблица за създаване на много към много връзка между таблиците „User" и „Listing". Тя съдържа и времеви полета за създаване и изтриване, като целта е запазените обяви да се извлича, сортирани по дата на създаване.

```
model UserSavedListings {
  uuid         String    @id @default(uuid())
  created_at   DateTime  @default(now())
  deleted_at   DateTime?
  user         User      @relation(fields: [user_uuid], references: [uuid])
  user_uuid    String
  listing      Listing   @relation(fields: [listing_uuid], references: [uuid])
  listing_uuid String

  @@unique([user_uuid, listing_uuid])
}
```

фиг. 2.55. – Модел на таблица „UserSavedListings"

2.3.18. Таблица “UserSavedPlaces”

Таблицата „UserSavedPlaces” е създадена по подобие на „UserSavedListings” само че за между таблиците „User” и „Place”.

```
model UserSavedPlaces {
  uuid       String    @id @default(uuid())
  created_at DateTime  @default(now())
  deleted_at DateTime?
  user       User      @relation(fields: [user_uuid], references: [uuid])
  user_uuid  String
  place      Place     @relation(fields: [place_uuid], references: [uuid])
  place_uuid String

  @@unique([user_uuid, place_uuid])
}
```

фиг. 2.56. – Модел на таблица „UserSavedPlaces”

2.3.19. Таблица “UserRating”

Таблицата „UserRating” е създадена по подобие на таблица „ListingRating“, но за оценяване на потребители от други потребители.

```
model UserRating {
  uuid            String    @id @default(uuid())
  rating          Int
  user_rated      User      @relation("user_rated", fields: [user_rated_uuid], references: [uuid])
  user_rated_uuid String
  created_at      DateTime  @default(now())
  deleted_at      DateTime?
  user            User      @relation("user", fields: [user_uuid], references: [uuid])
  user_uuid       String

  @@unique([user_uuid, user_rated_uuid])
}
```

фиг. 2.57. – Модел на таблица „UserRating”

2.3.20.Допълнителни таблици

Допълнителните таблици са извън заданието на дипломната работа и не се използват в текущата версия на софтуерния продукт. Те са създадени с цел за бъдещо развитие на проекта.

2.3.20.1. Таблица “PlaceReport”

```
model PlaceReport {
  uuid       String    @id @default(uuid())
  reason     String
  created_at DateTime  @default(now())
  deleted_at DateTime?
  user       User      @relation(fields: [userUuid], references: [uuid])
  userUuid   String
  place      Place     @relation(fields: [placeUuid], references: [uuid])
  placeUuid  String
}
```

фиг. 2.58. – Модел на допълнителна таблица „PlaceReport”

2.3.20.2. Таблица “ListingReport”

```
model ListingReport {
  uuid         String    @id @default(uuid())
  reason       String
  created_at   DateTime  @default(now())
  deleted_at   DateTime?
  user         User      @relation(fields: [user_uuid], references: [uuid])
  user_uuid    String
  listing      Listing   @relation(fields: [listing_uuid], references: [uuid])
  listing_uuid String
}
```

фиг. 2.59. – Модел на допълнителна таблица „ListingReport”

2.3.20.3. Таблица „UserSettings”

```
model UserSettings {
  uuid                  String    @id @default(uuid())
  preferred_theme       String    @default("light")
  marketing_emails      Boolean   @default(true)
  in_app_notifications Boolean   @default(true)
  email_notifications   Boolean   @default(true)
  created_at            DateTime  @default(now())
  updated_at            DateTime  @default(now()) @updatedAt
  deleted_at            DateTime?
  user                  User      @relation(fields: [user_uuid], references: [uuid])
  user_uuid             String    @unique
}
```

фиг. 2.60. – Модел на допълнителна таблица „UserSettings”

2.3.20.4. Таблица “Chat”

```
model Chat {
  uuid       String    @id @default(uuid())
  created_at DateTime  @default(now())
  updated_at DateTime  @default(now()) @updatedAt
  deleted_at DateTime?
  users      User[]    @relation("ChatUsers")
  messages   Message[]
}
```

фиг. 2.61. – Модел на допълнителна таблица „Chat”

2.3.20.5. Таблица “Message”

```
model Message {
  uuid       String    @id @default(uuid())
  content    String
  created_at DateTime  @default(now())
  updated_at DateTime  @default(now()) @updatedAt
  deleted_at DateTime?
  user       User      @relation(fields: [user_uuid], references: [uuid])
  user_uuid  String
  chat       Chat      @relation(fields: [chat_uuid], references: [uuid])
  chat_uuid  String
}
```

фиг. 2.62. – Модел на допълнителна таблица „Message”

2.3.21. Изброени типове

2.3.21.1. Изброен тип „Role“

Изборният тип "Role" се използва в таблицата с потребители за определяне на ролята и правата на даден потребител. В базата данни има 3 типа роли - обикновен потребител, модератор и администратор.

```
enum Role {
  USER
  MOD
  ADMIN
}
```

фиг. 2.63. - Модел на изброен тип "Role"

2.3.21.2. Изброен тип „State“

Изборният тип "State" се използва в таблицата с обявите за определяне на качеството и запазеността на дадено оборудване. В базата данни има 4 типа на състояние - ново, като ново, използвано и подновено / ремонтирано.

```
enum State {
  NEW
  LIKE_NEW
  USED
  REFURBISHED
}
```

фиг. 2.64. - Модел на изброен тип "State"

2.3.21.3. Изброен тип „Status“

Изброеният тип "Status" се използва в таблиците, които са за съдържание, създадено от потребител - обяви, места, снимки, тагове, коментари, услуги, ревюта, категории и услуги. Предназначението му е да се използва за статус на даден обект в базата данни - дали е публичен, или не. В базата данни има 4 типа статус - публичен, частен, премахнат и изтрит.

```
enum Status {
  PUBLIC
  PRIVATE
  REMOVED
  DELETED
}

```

фиг. 2.65. - Модел на изброен тип "Status"

2.3.21.4. Изброен тип „ListingType“

Изброеният тип "ListingType" се използва в таблицата за обяви за бъдещо развитие в предлагане в платформата не само на техника, но и на фрийланс[lxvii] услуги, свързвани с процеса по създаване на съдържание, от потребители. В базата данни има 2 типа обяви - обикновена обява за оборудване и услуга.

```
enum ListingType {
  PRODUCT
  SERVICE
}
```

фиг. 2.66. - Модел на изброен тип "ListingType"

Глава III

Реализация

Дипломната работа е логически разделена на две системи - сървърен софтуер (приложно-програмен интерфейс) и клиентски софтуер (потребителски интерфейс). Те са разгледани съответно в точки 3.1. и 3.2. в тази глава. Показаните извадки код са само малка част от дипломната работа, като са извадени само обобщени (опростени и уеднаквени) и по-специфични, нетривиални такива. Пълният изходен код на двата проекта може да бъдат разгледани като приложение, на електронен носител, на тази дипломна работа или в облачните хранилища в GitHub на следните интернет адреси:

- Сървърен софтуер - https://github.com/KokosTech/lenslend-backend
- Клиентски софтуер - https://github.com/KokosTech/lenslend-frontend

3.1. Сървърна част

3.1.1. Структура на файлова система

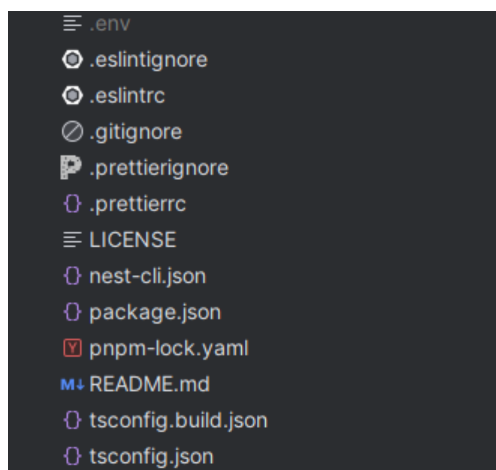
Конфигурационните файлове от един проект са изключително важна част от неговата цялост, благодарение на тях този проект е стандартизиран, изпълним и дефиниран за други системи, освен локалната.

- *.env* е конфигурационен файл, който се дефинира на всяка локална система и не се разпространява чрез VCS. Използва се за съхранение на конфигурационни променливи, които или са различни за всяка система, или са поверителни.
- *.eslintignore и .prettierignore* се използват за изброяване на пътища и файлове, които да бъдат игнорирани от

инструментите за статичен анализ и форматиране - ESLint и Prettier.

- *.eslintrc и .prettierrc* са конфигурационни файлове, в които се отбелязват използването на разширения, стандарти и промени в правилата на инструментите за статичен анализ и форматиране - ESLint и Prettier.
- *.gitignore*, по подобие на *.eslintignore* и *.prettierignore*, се използва за игнориране на пътища и файлове, но за тяхното следене чрез Git – контрол на версии.
- *.nest-cli.json* е автоматично генериран (от NestJS) конфигурационен файл за NestJS, който може да бъде променян от разработчика.
- *package.json* е конфигурационен файл за менажиране на зависимости с Node Package Manager (мениджър на зависимости за софтуер, базиран на Node.js). В него се конфигурират различни параметри на проекта като име, версия, описание, автор, лиценз, на каква версия на Node.js се изпълнява, както и различни скриптове, с които проекта разполага - примерно: стартиращ, компилиращ, форматиращ, тестващ и много други. В този файл са посочени и всички зависимости на проекта, заедно с техните версии, като отделно може да бъде посочен и специален софтуер за менажиране на зависимостите, заедно с неговата версия. В този проект се използва pnpm.
- *pnpm-lock.yaml* е автоматично генериран файл, който не трябва да бъде променян от разработчика. Използва се за следене на зависимостите от дадения мениджър на пакети. Подобни файлове има и от другите мениджъри - package-log.json (npm) и yarn.lock (yarn).

- *tsconfig.json и tsconfig.build.json* са конфигурационни файлове за TypeScript, неговите версии, правила и настройки за компилиране.



фиг. 3.1.1. Конфигурационни файлове във файловата структура на проекта

- *.github* - обикновено съдържа конфигурационни файлове и автоматизации за GitHub, но в този проект се използва за конфигуриране на шаблони за GitHub Issues и Pull Requests.
- prisma
    - *schema.prisma* - конфигурационен файл за Prisma ORM, който служи за дефиниране на моделите на базата данни и връзките между тях (разгледани в т. 2.3.). Този файл използва специфичен за Prisma синтаксис за описание на схемата на базата данни.
    - *migrations* - директория, в която се съхраняват всички миграции (генерирани от Prisma). Те съхраняват под формата на SQL файлове.
    - *seed.ts* - файл, в който са дефинирани записи по подразбиране (предефинирани), когато се създава нова база данни.
- *src* - директория, която съдържа целия изходен код на проекта. Структурата на повечето директории в нея е, че те са различни

функционалности (модули) на NestJS приложно-програмния интерфейс или обединение на няколко такива функционалности.

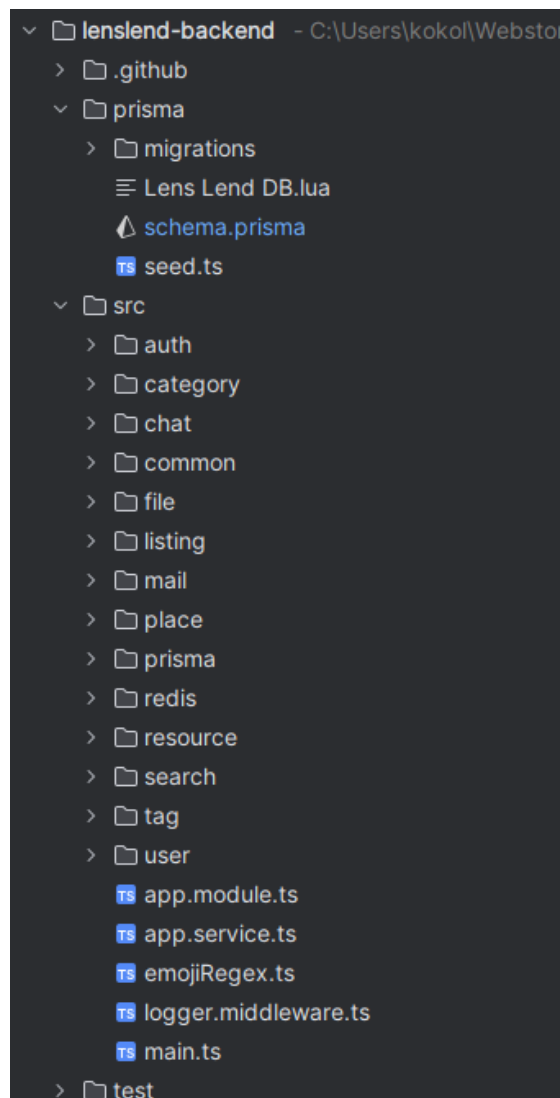
- *auth* - модул и функционалност за удостоверение и авторизация.
- *category* - модул и функционалност за категориите на места и обяви.
- *chat* - допълнителен модул и функционалност, която не е част от разработката на дипломната работа - чат.
- *common* - директория, в която се съхраняват споделени елементи, които се използват в проекта. Те могат да бъдат: декоратори, интерфейси, типове, грешки, филтри, интерсептори, dto, споделени помощни функции и други.
- *file* - модул и функционалност, която се използва за създаване и достъпване на обекти в AWS S3.
- *listing* - модул и функционалност за обяви и услуги. В него се съдържа модул и функционалност за създаване на коментари и сигнали за обяви и услуги.
- *mail* - допълнителен модул и функционалност, която не е част от разработката на дипломната работа - има за цел да бъде използван за изпращане на имейли за потвърждаване и изтриване на профил, както и нотификации при получаване на нови съобщения посредством вградената чат система.
- *place* - модул и функционалност за места. В него се съдържа модул и функционалност за създаване на ревюта, сигнали и услуги за места.
- *prisma* - модул за споделената бизнес логика и конфигурация - PrismaService, чрез която всички

елементи от бизнес логиката на приложение (Service) имат достъп до една инстанция на PrismaService чрез инжектиране на зависимостите (Dependency Injection). Използва се за правене на заявки до базата данни.

  - *redis* - модул за споделена бизнес логика и конфигурация - RedisService, изградена и използвана по подобен начин като PrismaService.
  - *resource* - модул и функционалност, която се използва вътрешно в приложение, за проверка на права на достъп до даден ресурс
  - *search* - модул и функционалност за търсене на потребители, обяви и места.
  - *tag* - модул за тагове, които се използват за обяви и места.
  - *user* - модул и функционалност за потребители. Тя разполага както със собствен контролер, така и се използва в модула за удостоверение и авторизация.
  - *app.module* - главният модулен файл на едно NestJS приложение.
  - *main.ts* - главният файл на едно NestJS приложение
- *test* - директория, в която се съдържат различни Unit тестове за проекта.

В повечето модули и функционалности се съдържат файлове и директории като: контролери, бизнес логика (service) - резолвър, множество dto, типове, селектори (за Prisma) и други. Те биват разгледани в по-долните точки в тази глава.

Структура и обобщение на файловата система могат да бъдат разгледани в изображението на *фиг. 3.1.2.* на следващата страница.



фиг. 3.1.2. - Директории и част от файловете в структурата на файловата система на проекта

3.1.2. Конфигуриране на връзка и достъп до базите данни - PostgreSQL с Prisma и Redis

За да може да се правят заявки към базата данни, трябва да се конфигурира и създаде инстанция на Prisma, която да се използва в приложението. Тази логика се нарича „Prisma Service", която унаследява класа PrismaClient, който е генериран от Prisma с „prisma generate" или на всяка миграция, направена с Prisma. Prisma Service имплементира и NestJS интерфейсите при инициализация и разрушаване на модула, в

които се извикват методите на PrismaClient за свързване и отвързване от базата данни.

```
@Injectable()
export class PrismaService
  extends PrismaClient
  implements OnModuleInit, OnModuleDestroy
{
  constructor() {
    super({
      log: ['info'],
    });
  }
  async onModuleInit() {
    await this.$connect();
  }

  async onModuleDestroy() {
    await this.$disconnect();
  }
}
```

фиг. 3.1.3. – Prisma Service

Заявките в Prisma се правят с извикване на *PrismaService* инстанция и метод на всеки модел като *findMany*, *findUnique*, findUniqueOrThrow и други. Тези методи приемат много различни аргументи, като един от тях е „select". Той се използва за избиране и вземане само на дадени полета от модел и неговите връзки. Тъй като *PrismaClient* е генериран спрямо модела, методите и техните параметри имат строга типизация, което подпомага автоматичното допълване, както и предотвратява грешки за избирани на несъществуващи полета и таблици.

```
const users = await prisma.user.findMany({
  select: {
    name: true,
    posts: {
      select: {
        title: true,
      },
    },
  },
})
```

фиг. 3.1.4. – Примерна заявка със select за избиране на полета

Това обаче създава един голям проблем и той е, че тези „select” параметри стават много големи, повторяеми и неподходящи за методи от бизнес логиката (service), затова за целта на този проект е създадена спомагателна функция, която се използва за създаването на типизирани select обекти в отделни файлове. Функцията е generic[lxviii], като тя приема модел от *PrismaClient*. Това позволява преизползването на select обекти, както и тяхното съхранение в отделни файлове и директории – „selects”.

```
import { Prisma } from '@prisma/client';

export type CheckSelectKeys<T, U> = {
  [K in keyof T]: K extends keyof U ? T[K] : never;
};

export const createSelect = <T extends Prisma.SelectSubset<unknown, unknown>>(
  arg: CheckSelectKeys<T, T>,
) => arg;
```

фиг. 3.1.5. – Подпомагаща функция createSelect()

На *фиг. 3.1.6.* е показан примерен select обект за избиране на снимка от обява. На функцията се подава модела, който е взет от *PrismaClient* за *ListingImage*.

```
import { Prisma } from '@prisma/client';
import { createSelect } from '../../common/utils/createSelect';

export const ListingImageSelect = createSelect<Prisma.ListingImageSelect>({
  uuid: true,
  url: true,
  alt: true,
  status: true,
});
```

фиг. 3.1.6. – Примерна употреба на createSelect()

Връзка с базата данни, допълнителни конфигурации и разширения се дефинират в prisma.schema файла.

```
generator client {
  provider        = "prisma-client-js"
  previewFeatures = ["fullTextSearch"]
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}
```

фиг. 3.1.7. – Връзка с базата данни, избор на нейния тип (PostgreSQL) и добавяне на функционалност за търсене (full text search)

За да може да се правят заявки до Redis базата данни е създаден клас с бизнес логика – *RedisService*. В него са създадени 3 метода, които са нужни за създаване, намиране и извличане на всички записи в базата данни. Тези методи са използвани в методите за удостоверение и авторизация.

Методът *setex()* приема три аргумента – ключ, стойност, които са стрингове, и време за живот на записа (ttl), което е число в секунди. В този метод се извиква *setEx()* метода на Redis клиента, който е инициализиран в RedisService.

```
async setex(key: string, value: string, ttl: number): Promise<void> {
  await this.client.setEx(key, ttl, value);
}
```

фиг. 3.1.8. – Метод setex() от RedisService

Методът get() приема само един аргумент – ключ, който е от тип стринг, като връща Promise от стринг или null стойност, ако е неуспешно изпълнен. В самия метод се изпълнява едноименната функция за Redis

клиента – get(), която приема същия тип аргумент. Неговата стойност се връща.

```
async get(key: string): Promise<string | null> {
  return this.client.get(key);
}
```

фиг. 3.1.9. – Метод get() от RedisService

Освен това в RedisService е създаден и метод за извличане на всички ключове, техните стойности и оставащ живот в един голям списък, наречен *getAll()*, който не приема никакви аргументи.

```
async getAll() {
  const keys = await this.client.keys('*');

  return Promise.all(
    keys.map(async (key) => ({
      token: key,
      user: await this.client.get(key),
      ttl: (await this.client.ttl(key)) || 0,
    })),
  );
}
```

фиг. 3.1.10. – Метод getAll() от RedisService

По подобие на *PrismaService*, RedisService е клас, създаден за достъп до Redis базата данни. Неговата конфигурация може да бъде видяна на *фиг. 3.1.11.*, която се намира на следващата страница. В нея се използва *ConfigService* – услуга, предоставена и вградена в NestJS, за извличане на данни от .env конфигурационен файл. Тя разполага с методите get и *getOrThrow*, които са силно типизирани. Като *getOrThrow* спомага, ако дадени конфигурационни стойности не са зададени, да не се стартира приложно-програмния интерфейс.

```
@Injectable()
export class RedisService implements OnModuleInit, OnModuleDestroy {
  private readonly client: redis.RedisClientType;

  constructor(private readonly config: ConfigService) {
    this.client = redis.createClient({
      url: this.config.getOrThrow<string>('REDIS_URL'),
      socket: {
        connectTimeout: 5000,
        noDelay: true,
        keepAlive: 1000,
        reconnectStrategy: (retries) => Math.min(retries * 50, 500),
      },
      username: this.config.getOrThrow<string>('REDIS_USERNAME'),
      password: this.config.getOrThrow<string>('REDIS_PASSWORD'),
      name: this.config.getOrThrow<string>('REDIS_NAME'),
      database: this.config.get<number>('REDIS_DATABASE', 0),
    });

    this.client.on('error', (error) => {
      ...
    });

    this.client.on('connect', () => {
      ...
    });
  }

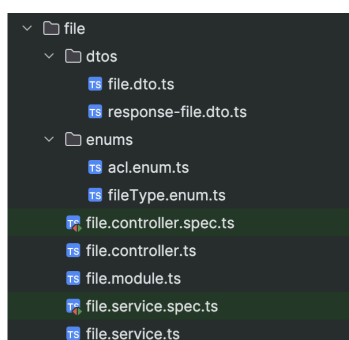
  async onModuleInit() {
    await this.client.connect();
    ...
  }

  async onModuleDestroy() {
    await this.client.disconnect();
    ...
  }
...
```

фиг. 3.1.11. – Конфигурация на RedisService и инициализация на Redis Client

3.1.3. Конфигуриране на услуга за качване на файлове

Проектът разполага с услуга на AWS S3 за качване и четене на файлове за обяви, места и потребители. В нея се съдържат както контролер, така и обслужващ слой, типове и няколко dto.



фиг. 3.1.12. – Файлова структура за функционалността за качване на файлове

В обслужващия слой *FileService* се инициализира AWS S3 клиента., като за него се задават регион и 2 ключа за достъп, които се взимат от *ConfigService*, при липсата им – приложението няма да стартира.

```
@Injectable()
export class FileService {
  private readonly s3Client: S3Client;

  constructor(private readonly config: ConfigService) {
    this.s3Client = new S3Client({
      region: this.config.getOrThrow<string>('AWS_S3_REGION'),
      credentials: {
        accessKeyId: this.config.getOrThrow<string>('AWS_ACCESS_KEY_ID'),
        secretAccessKey: this.config.getOrThrow<string>(
          'AWS_SECRET_ACCESS_KEY',
        ),
      },
    });
  }
...
```

фиг. 3.1.13. – Конструктор на FileService и инициализиране на S3

FileService съдържа само един метод – *uploadFile()*, който приема файлови параметри и потребителски идентификационен номер, като след това се създава уникален ключ (път) за файла, който ще бъде съхранен в S3 кофата. Този ключ се състои от главна директория, директория за потребителя, уникално генериран ключ и нормалното име на файла. След това се създава обект с параметри на обекта, в който се съдържат публичността му, името на кофата, ключа, типа на файла и метаданни – потребителски ключ, след което се изпълнява PUT команда и се взима готов URL адрес, който има изтичащ период от 5 минути за качване файла. След това методът връща URL за качване, ключа и публичен URL, който да се използва и съхранява в базата данни. Методът е описан във *фиг. 3.1.14.*, която се намира на следващата страница.

```
async uploadFile(
  fileParams: FileDto,
  userUuid: string,
): Promise<ResponseFileDto> {
  const { name, type, acl } = fileParams;
  const key = `${this.config.get('AWS_S3_FOLDER')}/${userUuid}/${uuid()}-${name}`;

  const params: PutObjectRequest = {
    Bucket: this.config.get<string>('AWS_S3_BUCKET_NAME'),
    Key: key,
    ACL: acl,
    ContentType: `image/${type}`,
    Metadata: {
      user_uuid: userUuid,
    },
  };

  const command = new PutObjectCommand(params);

  const url = await getSignedUrl(this.s3Client, command, {
    expiresIn: 60 * 5,
  });

  const public_url = `${this.config.get<string>('AWS_CLOUDFRONT_URL')}/${key}`;

  return {
    url,
    key,
    public_url,
  };
}
```

фиг. 3.1.14. – Метод uploadFile() на FileService

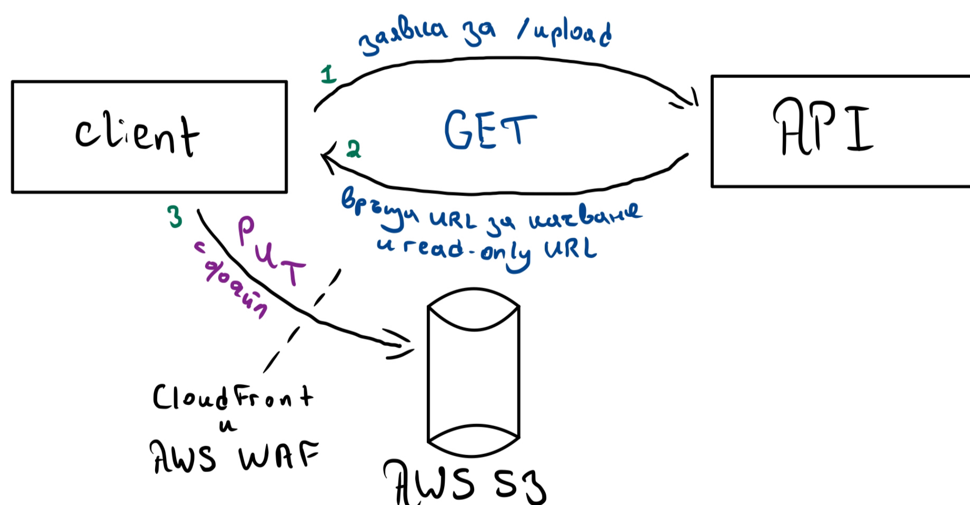
За качване на файл се използва контролер с път „/file/upload", който изисква потребителят да е с акаунт. Като се приемат заявки от URL адреса – име на файл, тип на файл и публичност.

```
@Controller('file')
@ApiTags('file')
@ApiBearerAuth()
@UseGuards(JwtAuthGuard)
export class FileController {
  constructor(private readonly fileService: FileService) {}

  @Get('upload')
  @ApiBearerAuth()
  @ApiResponse({
    status: 200,
    type: ResponseFileDto,
  })
  async uploadFile(
    @Req() req: RequestWithUser,
    @Query() query: FileDto,
  ): Promise<ResponseFileDto> {
    return this.fileService.uploadFile(query, req.user.uuid);
  }
}
```

фиг. 3.1.15. – Контролер за качване на файлове

Процесът за качване на файлове е описан на фиг. 3.1.16.



фиг. 3.1.16. – Процес по качване на файл

3.1.4. Обработване на ресурс, получен от интернет - проверка и верифициране на данни

Обяснението е общо и важи за всеки модул и функционалност, които получават данни от интернет независимо дали е посредством параметри, заявки или тела (json).

„Не може да се има доверие на потребителя". Всички данни, които са подадени от потребителя, минават през валидация. Тя най-често и лесно минава през „class-validator" pipes. Като тази библиотека се използва за всяко едно dto, което идва от потребителя (външния свят).

```
app.useGlobalPipes(
  new ValidationPipe({
    transform: true,
    whitelist: true,
    stopAtFirstError: true,
    exceptionFactory: (errors) => new ValidationException(errors),
  }),
);
```

фиг. 3.1.17. – Активиране на глобален Pipe за валидации

Най-добрият общ пример за такова dto е „CreatePlaceDto” *(фиг. 3.1.18.)*, което се използва за създаване на място. В него се съдържат всички данни, за да бъде създадено едно място, подадени от потребител.

За описание на изискванията за дадено поле се използват декоратори / анотации. В проекта са използвани част от следните, но има още много други, които могат да бъдат разгледани в документацията на Class-Validator.

- *@IsString()* – да е стринг
- *@IsNotEmpty()* – да не е празно
- *@Length(N, M)* – да е с дължина от N до M
- *@Matches(X, Y, Z)* – да съвпада на даден регулярен израз (Regex - X).
- *@IsUUID()* – да е UUID
- *@IsLatitude()* и *@IsLongitude()* – да са координати (в зависимост от посоченото)
- *@IsEnum(X)* – да съвпада с изброен тип X
- *@IsArray()* – да е масив
- *@ArrayNotEmpty()* – да не е празен масив
- *@ArrayMaxSize(X)* – да е масив с X максимален размер
- *@MinLength(X)* и *@MaxLength(X)* – да е с минимална или максимална дължина - X
- *@IsNumber()* – да е число
- *@Min(X)* и *@Max(X)* – да е число с минимална или максимална стойност - X
- *@IsPhoneNumber()* – да е валиден телефон

```
export class CreatePlaceDto {
  @ApiProperty()
  @IsString()
  @IsNotEmpty()
  @Length(6, 60)
  name: string;

  @ApiProperty()
  @IsString()
  @IsNotEmpty()
  @Matches(emojiRegex, '', {
    message: 'Invalid emoji',
  })
  icon: string;

  @ApiProperty()
  @IsString()
  @IsNotEmpty()
  @Length(100, 3000)
  description: string;

  @ApiProperty()
  @IsUUID()
  categoryUuid: string;

  @ApiProperty()
  @IsLatitude()
  @IsNotEmpty()
  lat: number;

  @ApiProperty()
  @IsLongitude()
  @IsNotEmpty()
  lng: number;

  @ApiProperty({
    enum: Status,
    default: 'PUBLIC',
  })
  @IsEnum(Status)
  @IsNotEmpty()
  status: Status;

  @ApiProperty()
  @IsArray()
  @IsString({ each: true })
  services: string[];

  @ApiProperty()
  @ArrayNotEmpty()
  @ArrayMaxSize(6)
  @IsString({ each: true })
  images: string[];

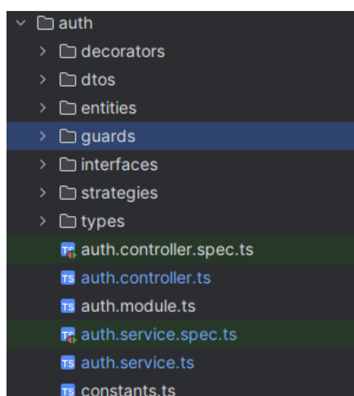
  @IsString({
    each: true,
  })
  @MinLength(3, {
    each: true,
  })
  @MaxLength(20, {
    each: true,
  })
  @ArrayNotEmpty()
  @ArrayMaxSize(16)
  @ApiProperty()
  tags: string[];
}
```

фиг. 3.1.18. - CreatePlaceDto

3.1.5. Създаване на потребителски профили, удостоверение, авторизация

Една от най-важните части на приложно-програмните интерфейси е сигурността – дали даден потребител е авторизиран, когато трябва, дали има права за даден обект и т.н.

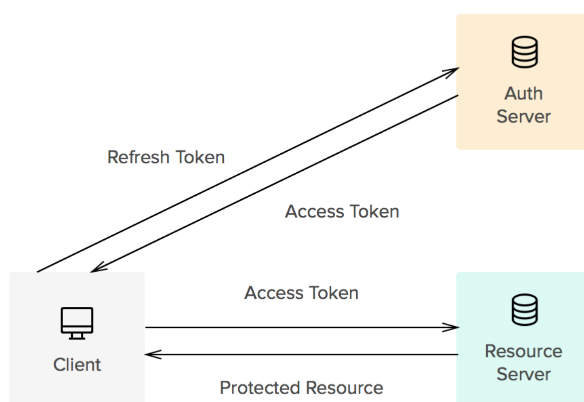
Структурата на модула за удостоверение и авторизация е следната: в директорията "decorators" се съдържат различни декоратори за управление на права, които са разгледани в т. 3.1.6., в "dtos" се съдържат шаблони за това, което сървъра получава от потребителя при влизане, регистриране и излизане. В "entity", "interfaces" и "types" се съдържат шаблони и типове, които се или връщат, или използват вътрешно в програмата. В "guards" се съдържат няколко NestJS guards, а техните стратегии - в "strategies". Този модул разполага с контролер и бизнес логика "service". "constants.ts" файлът съдържа JWT константи за модула.



фиг. 3.1.19 - Файлова структура на "Auth" функционалността

Както беше споменато в т. 2.2.13., в тази дипломна работа се използват JWT токени за авторизация. Подходът, който е взет, е да се използва т.нар. "JWT Refresh" стратегия. При успешно влизане (или регистриране на потребител) се връщат "access" и "refresh" токени. "access" токенът се използва за получаване на достъп до даден защитен ресурс и за потвърждаване (авторизиране) на самоличността. Тъй като

"access" токена се използва в изключително много заявки за авторизация, той подлежи на атаки и опити за кражба на достъп или самоличност. Поради това той е създаден да изтича за сравнително кратък период от време. И когато клиент се опита да извърши операция с изтекъл такъв токен, ще получи грешка, че е невалиден. Тогава клиентът трябва да се опита да направи т.нар. ротация и да поднови своя "access" токен. Тук идва и употребата на "refresh" токена, той се използва точно за това подновяване и само тогава. Поради нечестата му употреба "refresh" токена има сравнително по-дълъг живот спрямо "access" токена.



фиг. 3.1.20. - "access" - "refresh" токен стратегия

За изграждане на тази система в NestJS се използва разширената на Pasport.js библиотека - "NestJS Passport", и допълнителни функционалности от "passport-jwt" библиотеката. За тази цел са изградени три файла "Guard" и три стратегии.

Първата стратегия се използва само когато потребителят се опита да влезе в своя профил - това е локалната стратегия. Тя е разширение на паспортната стратегия от библиотеката. В нейния конструктор се уточнява, че полето за потребителско име е "email", вместо "username", което е по подразбиране. Задължителен метод за класове, които

разширяват *PassportStrategy*, е *validate()*, който се използва за валидиране. В локалната стратегия се използва метода от бизнес логиката за сигурност - *validateUser()*, която намира даден потребител по имейл адрес. При успешно намерен потребител, функцията връща имейла и паролата, а в обратния случай - null.

```
async validateUser(email: string, pass: string): Promise<unknown> {
  const user = await this.userService.findByEmail(email);

  if (user && (await compare(pass, user.password))) {
    return {
      email: user.email,
      password: user.password,
    };
  }

  return null;
}
```

фиг. 3.1.21. - validateUser()

При успешно валидиране на потребителя, стратегията връща върнатото от validateUser(), докато при неуспешно валидиране на данните на потребителя се връща грешка.

```
@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy) {
  constructor(private authService: AuthService) {
    super({ usernameField: 'email' });
  }

  async validate(username: string, password: string) {
    const user = await this.authService.validateUser(username, password);

    if (!user) {
      throw new UnauthorizedException();
    }

    return user;
  }
}
```

фиг. 3.1.22. - Локална стратегия

Стратегиите се използват при създаването на NestJS Guard. Като в този случай локалната стратегия се използва в локален guard -

"LocalAuthGuard", който разширява класа "AuthGuard", за да използва "local" стратегия.

```
@Injectable()
export class LocalAuthGuard extends AuthGuard('local') {}
```

фиг. 3.1.23. - Локален Guard

Този guard се използва само за вход на потребител в приложението в контролера за сигурност - "/auth/login". За използването на един или повече guard се употребява анотацията "UseGuards" над метода на пътя.

```
@Post('login')
@UseGuards(LocalAuthGuard)
@ApiOkResponse({
  type: AuthEntity,
})
async login(@Body() { email, password }: LoginDto) {
  return this.authService.login(email, password);
}
```

фиг. 3.1.24. - Метод login() на пътя "/auth/login" за Auth функционалността

Бизнес логиката зад пътя се намира в метода *login()* - *фиг. 3.1.25.* и *3.1.26.*. В него се проверяват съществуването на потребителя и се сравняват паролите. Като при сбъркани данни се връщат грешки.

```
async login(email: string, password: string) {
    const user = await this.userService.findByEmail(email);

    if (!user) {
      throw new NotFoundException('NO_USER_FOUND');
    }

    if (!(await compare(password, user.password))) {
      throw new UnauthorizedException('INVALID_PASSWORD');
    }

    ...
  }
```

фиг. 3.1.25. - Проверки в метода login()

След това се връщат двата токена - "access" и "refresh", Като информацията която се съдържа в "access" токена е потребителския идентификационен номер, докато в "refresh" токена се съдържат както и ИН, така и ИН на самия токен. Те се кодират с тайните ключове (един за "access" токена и друг за "refresh" токена), които се пазят в .env конфигурацията на сървъра. Задават се и техните животи, които също са указани в конфигурационния файл (дължината на живот и тайният ключ са описани в Auth модула при регистриране на JWT услугата).

```
return {
  access_token: this.jwtService.sign({
    id: user.uuid,
  }),
  refresh_token: this.jwtService.sign(
    {
      id: user.uuid,
      tokenId: uuid(),
    },
    {
      secret: jwtConstants.refreshSecret,
      expiresIn: jwtConstants.refreshExpiresIn,
    },
  ),
};
```

фиг. 3.1.26. - Създаване и връщане на токените в login()

Следващата стратегия е JWT стратегията. Тя се използва навсякъде, където са нужни данните на потребителя и се изисква потребителят да бъде с профил. Тази стратегия също разширява PassportStrategy класа, като приема модифицирано име - "jwt". В нейния конструктор се определя откъде се взима JWT токена - от Auth хедъра на заявката, какъв е тайният ключ на сървъра, зареден от .env конфигурационния файл и дали да се игнорират изтекли токени, като в случая на този проект се игнорират, за да в Guard да се върне специална (custom) грешка. В задължителната validate() се проверява съществуването на потребителя спрямо идентификационния номер на потребителя, който се съдържа в JWT токена. При ненамерен потребител се връща грешка, а в обратния

случай се предават данните на потребителя към методите в контролерите под формата част от заявката.

```
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy, 'jwt') {
  constructor(private userService: UserService) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: jwtConstants.secret,
      ignoreExpiration: false,
    });
  }

  async validate(payload: { id: string }) {
    const user = await this.userService.findByUUID(payload.id);

    if (!user) {
      console.warn(payload);
      throw new UnauthorizedException();
    }

    return user;
  }
}
```

фиг. 3.1.27. - JWT стратегия

JWT Guard, който разширява AuthGuard с персонализираната стратегия - JWT, в него единствено се прави проверка за невалиден (изтекъл) токен, ако той е такъв, се връща специална грешка за изтекъл токен. Този Guard може да се използва с анотацията @UseGuards(JwtAuthGuard).

```
@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
  handleRequest(
    err: any,
    user: any,
    info: any,
    context: ExecutionContext,
    status: any,
  ) {
    if (info instanceof Error) {
      throw new UnauthorizedException({
        errorCode: 'INVALID_TOKEN',
      });
    }

    return super.handleRequest(err, user, info, context, status);
  }
}
```

фиг. 3.1.28. - JWT Guard

Последната стратегия, която се използва за токените се казва "Refresh JWT Strategy". Тя, по подобие на локалната стратегия, се използва само на едно място, но то е пътя за подновяване на "access" токена от "refresh" токена. В тази стратегия само се конфигурира PassportStrategy класа и се връщат данните, съхранени в токена - ИН на токен и потребител. Тази стратегия се използва в сходен (с изключението на персонализираното име, което тук е "jwt-refresh") guard, който може да бъде използван чрез анотацията @UseGuards(RefreshJwtGuard)

```
@Injectable()
export class RefreshJwtStrategy extends PassportStrategy(
  Strategy,
  'jwt-refresh',
) {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: jwtConstants.refreshSecret,
      ignoreExpiration: false,
    });
  }

  validate(payload: { id: string; tokenId: string }) {
    return {
      id: payload.id,
      tokenId: payload.tokenId,
    };
  }
}
```

фиг. 3.1.29. - Refresh JWT стратегия

Подновяването на токена се извършва чрез пътя "/auth/refresh". Този път е защитен с RefreshJwtGuard.

```
@Get('refresh')
@UseGuards(RefreshJwtGuard)
@ApiOkResponse({
  type: AuthEntity,
})
async refreshToken(@Request() req: RequestWithTokenInterface) {
  return this.authService.refreshTokens(req.user.id, req.user.tokenId);
}
```

фиг. 3.1.30. - Път за подновяване на токени - "/auth/refresh"

Бизнес логиката зад този път се намира в метода refreshTokens() - *фиг. 3.1.31.*, която приема данните от токена - ИН на потребителя и самия "refresh" токен. В метода се правят допълнителни проверки дали ИН съществуват и дали токенът е валиден, сравнен със списък от Redis базата данни. Ако такъв токен съществува в черния списък, означава, че токенът е невалиден и се връща грешка. Повече информация за черния списък на токени и неговата логика може да бъдат намерени по-надолу в тази точка при обясненията за излизане на потребител. След това се прави заявка към главната база данни, в която се търси потребител по идентификационен номер, ако не бъде открит, се връща грешка. В обратния случай, се създава и връща нов, подновен "access" токен.

```
async refreshTokens(id: string, tokenId: string) {
  if (!id) {
    throw new UnauthorizedException('NO_TOKEN_PROVIDED');
  }

  const isTokenIdValid = await this.redisService.get(tokenId);
  if (isTokenIdValid) {
    throw new UnauthorizedException('INVALID_REFRESH_TOKEN');
  }

  const user = await this.userService.findByUUID(id);

  if (!user) {
    throw new NotFoundException('USER_NOT_FOUND');
  }

  return {
    access_token: this.jwtService.sign({
      id: user.uuid,
    }),
  };
}
```

фиг. 3.1.31. - Метод refreshTokens()

Тъй като на клиентския софтуер, регистрацията е многостъпкова, има три пътя за регистрация - първите два са само за валидация и потвърждаване на данни, както и проверка за конфликтни записи (дали даден имейл, телефон или потребителско име вече съществуват, както и

за проверка на парола и потвърждаваща парола). Те са разделени на две стъпки - профилна (първа стъпка) и персонална (втора стъпка) информация.

```
@Post('validate/account')
async validateAccountData(@Body() body: SignupOneDto) {
  return this.authService.validateSignup(body, 1);
}

@Post('validate/personal')
async validatePersonalData(@Body() body: SignupDto) {
  return this.authService.validateSignup(body, 2);
}
```

фиг. 3.1.32. - Пътища за валидиране на данни и намиране на конфликти в многостъпкова регистрация

```
async validateSignup(body: SignupDto | SignupOneDto, step: number) {
  if (body.password !== body.confirmPassword) {
    throw new BadRequestException({
      message: 'passwordConfirm',
      code: 'PASSWORDS_DO_NOT_MATCH',
    });
  }

  const email = await this.userService.findByEmail(body.email);

  if (email) {
    throw new ConflictException({
      message: 'email',
      code: 'EMAIL_ALREADY_IN_USE',
    });
  }

  const username = await this.userService.findByUsername(body.username);

  if (username) {
    throw new ConflictException({
      message: 'username',
      code: 'USERNAME_ALREADY_IN_USE',
    });
  }

  if (step === 2 && body instanceof SignupDto) {
    const phone = await this.userService.findByPhone(body.phone);

    if (phone) {
      throw new ConflictException({
        message: 'phone',
        code: 'PHONE_ALREADY_IN_USE',
      });
    }
  }
}
```

фиг. 3.1.33. - Проверка за конфликти на данни по време на регистрация

Третият път отговаря за самата регистрация - "/auth/signup". Данните за регистрация се приемат в тялото на заявката, а те биват: имена, дата на раждане, телефон, имейл адрес, потребителско име, парола и потвърждаваща парола.

```
@Post('signup')
@ApiOkResponse({
  type: AuthEntity,
})
async signup(@Body() body: SignupDto) {
  return this.authService.signup(body);
}
```

фиг. 3.1.34. - Път за регистрация на потребител - "/auth/signup"

Самата логика се намира в метода signup(), който е сравнително кратък, поради факта, че повечето логика за валидиране на данни е изнесена в горепосочения метод във *фиг. 3.1.33.*

```
async signup(body: SignupDto) {
  await this.validateSignup(body, 2);

  const newUser = plainToClass(CreateUserDto, body);
  newUser.name = `${body.firstName} ${body.lastName}`;
  newUser.date_of_birth = new Date(body.dateOfBirth).toISOString();

  await this.userService.createUser(newUser);

  return this.login(body.email, body.password);
}
```

фиг. 3.1.35. - Метод signup() за регистрация на потребители

Проблемът с излизането на даден потребител, използвайки текущата JWT Refresh технология без запазване на състоянието или използване на сесии, е, че "refresh" токените остават валидни дори след излизане, което не е ефект, който искаме, особено когато "refresh" токените са с дълга валидност от 7 до 30 дена. Това означава, че ако потребител излезе от дадено устройство, но си запази "refresh" токена,

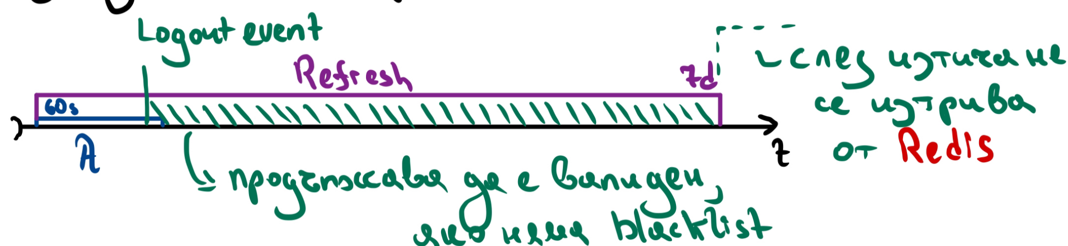
може след това да го използва, за да си издаде "access" токен. Това е потенциален метод за атака, който не трябва да се позволява.



фиг. 3.1.36. - Разлика между времето на живот на "access" и "refresh" токените

В тази дипломна работа се използва т.нар. "черен списък" за съхранение на "refresh" токени, който цели да се съхрани простотата и бързината на системата без запазване на състоянието на потребителя в базата данни. Както е споменато в т. 2.2.14., за създаването на този черен списък се използва Redis, поради нейната бързина и възможност за задаване на време на живот на записите, подобно на бисквитките и JWT токените.

В този черен списък се съдържат всички "refresh" токени, които все още не са изтекли. Веднага след тяхното изтичане (когато ttl стане равно на 0), Redis базата данни автоматично премахва записите, което намалява сравнително сложността и мястото, което се използва.



фиг. 3.1.37. - Логика зад създаването на черен списък и съхранението на "refresh" токени, които изтичат след примерните 7 дни

Излизането на потребител става чрез пътя "/auth/logout", като при него трябва да подаде както и двата токена в тялото на заявката, така и да подаде "refresh" токена като Authorization хедър в заявката.

```
@Post('logout')
@UseGuards(RefreshJwtGuard)
@ApiBearerAuth('refresh-token')
@ApiOkResponse({
  type: AuthEntity,
})
async logout(
  @Body()
  { accessToken, refreshToken }: LogoutDto,
) {
  return this.authService.logout(accessToken, refreshToken);
}
```

фиг. 3.1.38. - Път за излизане на потребител - "/auth/logout"

Бизнес логиката на този път се намира в logout() метода на Auth Service. В нея първо се извличат данните от "refresh" токена - ИН на потребителя, токена и датата и часа му на изтичане. Проверява се дали потребителят съществува и дали "refresh" токенът е валиден и не се намира в черния списък. Ако всичко това е изпълнено, токенът се записва в Redis базата данни, заедно с неговия оставащ живот.

```
async logout(accessToken: string, refreshToken: string) {
  const { id, tokenId, exp }: RefreshTokenInterface =
    await this.jwtService.verifyAsync(refreshToken, {
      secret: process.env.JWT_REFRESH_SECRET,
    });

  const user = await this.userService.findByUUID(id);

  if (!user) {
    throw new NotFoundException('USER_NOT_FOUND');
  }

  const isTokenIdInvalid = await this.redisService.get(tokenId);

  if (isTokenIdInvalid) {
    throw new UnauthorizedException('INVALID_REFRESH_TOKEN');
  }

  const expiresInSeconds = exp - Math.floor(Date.now() / 1000) + 1;
  await this.redisService.setex(tokenId, id, expiresInSeconds);

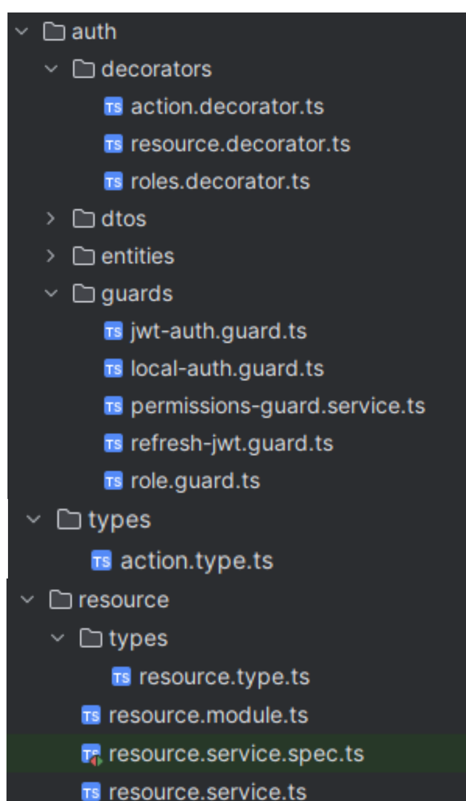
  return {
    access_token: accessToken,
    refresh_token: refreshToken,
  };
}
```

фиг. 3.1.39. - Метод logout()

3.1.6. Права за достъп и модифициране

Както беше споменато в т. 3.1.5., една от най-важните части на един ППИ е сигурността. В тази точка се разглежда втория аспект от сигурността – дали даден потребител има права за достъпване и/ли модифициране на даден обект.

С цел създаване на система за роли и права, в приложно-програмния интерфейс е създадена бизнес логика за взимане на собственик на даден обект - обява, място, ревю или коментар. Както и два NestJS Guards - Role Guard и Permissions Guard. Заедно с декоратори.



фиг. 3.1.40. Файлове, използвани за права за достъп и модифициране

Декораторите са за изискване на роля или поставяне на изискване за даден ресурс и определено действие.

Декораторът за роли приема тип роли, които са със стойности - "admin" (администратор, който може да прави всичко), "mod" (модератор,

който може да редактира или изтрива публикации) и "user" (нормален потребител, който може редактира или изтрива само свои публикации).

```
export const Roles = (...args: Role[]) => SetMetadata('roles', args);
```

фиг. 3.1.41. - Декоратор за роли - Role

Декораторът *@Action* се използва за означаване на даден път какво действие извършва. Този декоратор се използва от услугата за разрешения. Действията могат да бъдат - "view" (да може да се гледа дадена публикация) и "manage" (да може да се редактира или изтрива дадена публикация).

```
export const Action = (...args: ('view' | 'manage')[]) =>
  SetMetadata('action', args);
```

фиг. 3.1.42. - Декоратор за вид действие - Action

Последният декоратор, който се използва за създаване на тази функционалност е свързан с предишния - Action. Ресурсния декоратор (Resource) се използва за посочване на какъв ресурс се изпълнява дадено действие. Трябва да се настрои, тъй като трябва да се знае в коя таблица трябва да се търси даден уникален ключ. Стойностите, които могат да бъдат използвани са: "listing" (обяви и услуги), "place" (места), "comment" (коментари) и "review" (ревюта).

```
export const Resource = (
  ...args: ('listing' | 'place' | 'comment' | 'review')[]
) => SetMetadata('resource', args);
```

фиг. 3.1.43. - Декоратор за вид ресурс - Resource

Горепосоченият декоратор за ресурси се използва с цел уточняване на дадения ресурс, който се използва в бизнес логиката за ресурси. Тя зависи от всички останали резолвъри (Service), които представляват даден ресурс, генериран от потребителя. Има само един метод, който получава ИН на дадения ресурс и неговия тип. Базирано на това извиква заявка към базата данни чрез дадения резолвър.

```
@Injectable()
export class ResourceService {
  constructor(
    private readonly listingService: ListingService,
    private readonly placeService: PlaceService,
    private readonly commentService: CommentService,
    private readonly reviewService: ReviewService,
  ) {}

  async findOneMeta(
    resourceType: ResourceType,
    id: string,
  ): Promise<ResourceContent | null> {
    switch (resourceType) {
      case 'listing':
        return this.listingService.findOneMeta(id);
      case 'place':
        return this.placeService.findOneMeta(id);
      case 'review':
        return this.reviewService.findOneMeta(id);
      case 'comment':
        return this.commentService.findOneMeta(id);
    }
  }
}
```

фиг. 3.1.44. - Бизнес логика за взимане на метаданни за ресурси

Услугите на този метод се използват от Permissions Guard, който имплементира интерфейса CanActivate, който допуска или не допуска изпълнението на методи от пътища. Този Guard зависи от вградения Reflector клас и от Auth и Resource услугите.

```
@Injectable()
export class PermissionsGuard implements CanActivate {
  constructor(
    private readonly reflector: Reflector,
    private readonly authService: AuthService,
    private readonly resourceService: ResourceService,
  ) {}

  ...

}
```

фиг. 3.1.45. - Начална част от Permissions Guard

Създадени са няколко помощни метода за Permissions Guard, най-главни сред тях бивайки - getObjectOwnerAndStatus() и can(). getObjectOwnerAndStatus() се използва за взимане на метаданните за обект. От рефлектора, който взима информация от setMetadata() - метаданни, приложени заедно с анотацията @Resource, за типа ресурс, като след това избира кое UUID да използва за идентификатор на ресурс - uuid, ако е главен ресурс като обяви и места, или sub_uuid, ако е подресурс - коментари и ревюта. След това връща резултата на findOneMeta() метода от услугата за ресурси.

```
private async getObjectOwnerAndStatus(
  context: ExecutionContext,
): Promise<ResourceContent | null> {
  const [resource] = this.reflector.get<ResourceType[]>(
    'resource',
    context.getHandler(),
  );

  const request: RequestWithParamUuid = context.switchToHttp().getRequest();
  const resourceId = ['review', 'comment'].includes(resource)
    ? request.params.sub_uuid ?? request.params.uuid
    : request.params.uuid;

  return this.resourceService.findOneMeta(resource, resourceId);
}
```

фиг. 3.1.46. - Метод getObjectOwnerAndStatus()
за изземане на метаданни на ресурс

Преди да бъде разгледан *can()* метода. Той зависи от помощен метод - *getUserFromRequest()*. Метод, който проверява дали се е извършила вече проверка от *JWT Guard* на приложението (съответно ако пътят е защитен). Ако не е, се очаква да се изземе от токена и да се направят проверки за валидност чрез *getUserFromToken()*.

```
private async getUserFromRequest(
  request: Request | RequestWithUser,
): Promise<User | null> {
  if ('user' in request && request.user && 'uuid' in request.user) {
    return request.user;
  }

  const token = getJwtToken(request as Request);
  return this.authService.getUserFromToken(token);
}
```

фиг. 3.1.47. - Помощен метод getUserFromRequest() за изземан на потребител от заявка (независимо дали е минала през JWT Guard)

getUserFromToken() е спомагателен метод, но от Auth Service. В него се извлича ИН на потребителя и след това сравнява с базата данни - или ще върне null, ако такъв потребител не съществува, или ще върне цялата информация на потребителя, по подобие на JWT Guard.

```
async getUserFromToken(token: string | null): Promise<User | null> {
  if (!token) {
    return null;
  }

  try {
    const { id }: AuthTokenInterface = await this.jwtService.verifyAsync(
      token,
      {
        secret: jwtConstants.secret,
      },
    );

    return this.userService.findByUUID(id);
  } catch {
    return null;
  }
}
```

фиг. 3.1.48. - Помощен метод getUserFromToken() за извличане на потребител от JWT "access" токен

Помощният метод *can()* взима както и контекст, така и тип на метаданни на ресурс (ИН, ИН на създател и статус на публикацията). Взима потребителя от заявката и ако няма такъв, връща грешка, че клиента няма права. След това проверява дали потребителят е модератор или администратор, ако е - връща истина. Ако не е, се извиква помощна функция - *matchOwner()*, която сравнява ИН на потребителя и този на създателя на дадената публикация.

```
private async can(
  context: ExecutionContext,
  resource: ResourceContent,
): Promise<boolean> {
  const request: Request = context.switchToHttp().getRequest();
  const user = await this.getUserFromRequest(request);

  if (!user) {
    throw new UnauthorizedException(`Cannot manage object ${resource.uuid}`);
  }

  if (inArray([Role.ADMIN, Role.MOD], user.role)) {
    return true;
  }

  return this.matchOwner(user.uuid, resource.ownerId);
}
```

фиг. 3.1.49. - Метод can() за преценяване дали даден потребител има права над ресурс

canActivate() е булеви метод, който отговаря дали изпълнението на заявката може да продължи, или не. Това е метод, който се имплементира от интерфейса. Чрез рефлектора за метаданни се определя даденото действие, което потребителя иска да изпълни. Изземат се метаданните на обекта чрез спомагателния метод - *getObjectOwnerAndStatus()* и се прави проверка дали такъв обект съществува. След това, в зависимост от избраното действие:

- менажиране - веднага се извиква спомагателния метод *can()*.
- гледане - първо се проверява дали дадената публикация е публична, ако е - всички потребители се допускат, ако не е - извиква се спомагателния метод *can()*.

```
async canActivate(context: ExecutionContext): Promise<boolean> {
  const [action] = this.reflector.get<ActionType[]>(
    'action',
    context.getHandler(),
  );

  if (!action) {
    return false;
  }

  const resource = await this.getObjectOwnerAndStatus(context);
  if (!resource) {
    throw new NotFoundException('Resource not found');
  }

  switch (action) {
    case 'manage':
      return this.can(context, resource);
    case 'view':
      if (resource.status === Status.PUBLIC) return true;
      return this.can(context, resource);
    default:
      return false;
  }
}
```

фиг. 3.1.50. - задължителен булеви метод canActivate() за определяне дали дадена заявка може да продължи или не

Последният клас (и Guard) за справяне със сигурността е *Role Guard*, който се използва за лимитиране на достъпа до пътища само до потребители с дадена роля. При използването на този Guard, извикването на *JWT Auth Guard* е задължително, тъй като се сравняват роли на вече съществуващ потребител в заявката. След изземането на

потребителя, се проверява дали ролята му присъства в списъка (подаден от *setMetadata()*, който е направен в декоратора *Roles*), ако е - заявката продължава, в противния случай - приключва с грешка за авторизация и липса на права.

```
@Injectable()
export class RoleGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  matchRoles(roles: string[], userRole: string) {
    return roles.some((role) => role === userRole);
  }

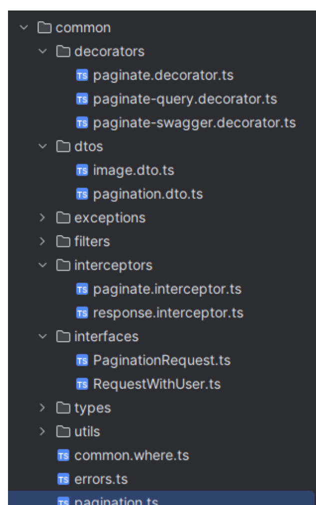
  canActivate(context: ExecutionContext): boolean {
    const roles = this.reflector.get<string[]>('roles', context.getHandler());
    if (!roles) {
      return true;
    }
    const request: RequestWithUser = context.switchToHttp().getRequest();
    const { user } = request;

    return this.matchRoles(roles, user.role);
  }
}
```

фиг. 3.1.51. - Role Guard за допускане само на дадени роли

3.1.7. Извличане на данни по страници (Pagination)

Една задължителна функционалност, която модерните приложно-програмни интерфейси трябва да имат, е извличането на дадени данни по страници (Pagination)



фиг. 3.1.52. - Файлова структура на "common", където се държат част от файловете за Pagination

За разработката са създадени набор от типове, интерфейси, декоратори, както dto и интерсептор на отговор, които се намират в “common” директорията.

Интерсепторът трансформира отговора, да следва стандарта на Pagination. Ако данните вече са от тип Pagination Result Dto, следва че те ще бъдат върнати такива, ако пък не са, то те ще бъдат трансформирани в такива.

```
@Injectable()
export class TransformInterceptor<T>
  implements NestInterceptor<T, PaginationResultDto<T>>
{
  intercept(
    context: ExecutionContext,
    next: CallHandler<T>,
  ): Observable<PaginationResultDto<T>> {
    return next.handle().pipe(
      map((data): PaginationResultDto<T> => {
        if (data instanceof PaginationResultDto) {
          return data as PaginationResultDto<T>;
        }

        const items: T[] = Array.isArray(data) ? data : [data as unknown as T];
        return new PaginationResultDto<T>(items, 1, 10, items.length);
      }),
    );
  }
}
```

фиг. 3.1.53. - Трансформиращ интерсептор за Pagination

Като резултатът, който се получава при всеки маршрут, който работи със страници, е следният: данни от общ тип, номер на страница (page), брой елементи на страница (limit) и общо колко елемента има (totalCount).

```
export class PaginationResultDto<T> {
  data: T[];
  @ApiProperty()
  page: number;
  @ApiProperty()
  limit: number;
  @ApiProperty()
  totalCount: number;

  constructor(data: T[], page: number, limit: number, totalCount: number) {
    this.data = data;
    this.page = page;
    this.limit = limit;
    this.totalCount = totalCount;
  }
}
```

фиг. 3.1.54. - Шаблон за отговор със страници

В заявката, която се подава от клиента се съдържат следните параметри: номер на страница и брой елементи на страница.

```
export interface PaginationRequest extends Request {
  query: {
    page: string;
    limit: string;
  };
}
```

фиг. 3.1.55. - Заявка с разширение да поддържа аргументи за страница

За да се улесни употребата на тези параметри, са създадени декоратори, които автоматично ги извличат в променливи, които връщат клас Pagination, съдържащ номер на страницата и бройката елементи на страница.

```
export const Paginate = createParamDecorator(
  (data: unknown, ctx: ExecutionContext): Pagination => {
    const request: PaginationRequest = ctx.switchToHttp().getRequest();
    const page = parseInt(request.query.page, 10) || 1;
    const limit = parseInt(request.query.limit, 10) || 10;
    return new Pagination(page, limit);
  },
);
```

фиг. 3.1.56. - Декоратор за извличане на параметри

Освен този декоратор, са създадени още два с цел помощ за пълна и правилна документация на Open-API (която може да бъде посетена на "{адрес на сървъра}/api") или още познато като Swagger. Единият декоратор е за параметрите - страница и лимит, а другият е за връщането на отговор.

```
export const ApiQueryPaginated = (page = 1, limit = 10) =>
  applyDecorators(
    ApiQuery({
      name: 'page',
      required: false,
      example: page,
    }),
    ApiQuery({
      name: 'limit',
      required: false,
      example: limit,
    }),
  );

```

фиг. 3.1.57. - Декоратор за документирани ППИ параметри за страници

```
export const ApiOkResponsePaginated = <DataDto extends Type<unknown>>(
  dataDto: DataDto,
) =>
  applyDecorators(
    ApiExtraModels(PaginationResultDto, dataDto),
    ApiOkResponse({
      schema: {
        allOf: [
          { $ref: getSchemaPath(PaginationResultDto) },
          {
            properties: {
              data: {
                type: 'array',
                items: { $ref: getSchemaPath(dataDto) },
              },
            },
          },
        ],
      },
    }),
  );
```

фиг. 3.1.58. - Декоратор за документиран ППИ отговор за страница

Така вече приложно-програмният интерфейс разполага с готов интерфейс и преизползваем код (само с няколко анотации) за разделянето на големите отговори по страници и намаляване / увеличаване на техния размер. Повечето примери за Pagination (поставяне на страници и лимити) могат да бъдат разгледани в CRUD ("Операции на ...") точките в тази глава. Общо казано се добавят към Prisma заявките take (лимит) и skip (колко да бъдат пропуснати - изчисление между текуща страница и лимит).

3.1.8. Обработване на грешки и форматиране на отговор

Както беше обяснено в т. 2.2.6., в NestJS филтрите се използват за обработка на грешки. В този приложно-програмен интерфейс трябва да се обработват 3 основни типа грешки - от валидация (след неспазване на оказан тип в DTO), от Prisma ORM (при грешки в заявките - несъществуващ запис, липса на параметри, конфликти и други), както и всички останали грешки, които или са програмирани да бъдат връщани (в изходния код), или неочаквани.

Най-простият филтър е *PrismaClientExceptionFilter*, който има за цел да събира и обработва грешките от тип *PrismaClientKnownRequestError*. Създадени са прости условия за проверка на кода на грешка, който Prisma връща. Всички тези кодове са описани в документацията на Prisma ORM, като за нуждите на този проект се използват двете най-често срещани грешки - *P2002* (конфликт) и *P2025* (не е намерен запис - връща се най-често при извикването на *findUniqueOrThrow*).

```
@Catch(Prisma.PrismaClientKnownRequestError)
export class PrismaClientExceptionFilter extends BaseExceptionFilter {
  catch(exception: Prisma.PrismaClientKnownRequestError, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const message = exception.message.replace(/\n/g, '');

    const cause = exception.meta?.cause;

    switch (exception.code) {
      case 'P2002': {
        const status = HttpStatus.CONFLICT;
        response.status(status).json({
          code: status,
          message,
          data: null,
          errors: null,
        });
        break;
      }
      case 'P2025': {
        const status = HttpStatus.NOT_FOUND;
        response.status(status).json({
          code: 'NOT_FOUND',
          message: cause,
          data: null,
          errors: null,
        });
        break;
      }
    }
  }
}
```

фиг. 3.1.59. - Филтър за обработване на Prisma грешки

Следващият филтър е валидационният, той връща всички грешки, които class-validator му е върнал.

```
@Catch(ValidationException)
export class ValidationExceptionFilter implements ExceptionFilter {
  private readonly i18nCommonErrorsFile = 'common-errors';

  catch(exception: ValidationException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();

    const { errors } = exception;

    console.log(exception);

    response.status(HttpStatus.BAD_REQUEST).json({
      code: 'VALIDATION_ERROR',
      message: 'Validation Error',
      data: null,
      errors,
    });
  }
}
```

фиг. 3.1.60. - Филтър за обработване на валидационни грешки

Главният филтър за обработване на грешки е Exception Filter. Той имплементира Exception Filter на NestJS, като обработва нормалните грешки и ги връща, или при фатални такива, връща не само отговор, но и запазва в лог файловете си и терминала.

```
@Catch()
export class ExceptionsFilter implements _ExceptionFilter {
  private readonly logger = new Logger('ExceptionHandler');

  catch(exception: unknown, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();

    let code: string;
    let message: string;
    let statusCode: HttpStatus;

    if (exception instanceof NotFoundException) {
      statusCode = HttpStatus.NOT_FOUND;
      code = 'NOT_FOUND';
      message = 'Not Found';
    } else if (exception instanceof HttpException) {
      statusCode = exception.getStatus();
      const resp = exception.getResponse() as HttpExceptionBody & {
        code: string;
      };

      // eslint-disable-next-line prefer-destructuring
      code = resp.code;
      message = resp.message as string;
    } else {
      statusCode = HttpStatus.INTERNAL_SERVER_ERROR;
      code = 'INTERNAL_SERVER_ERROR';
      message = 'Internal Server Error';

      if (exception instanceof Error) {
        this.logger.error(`${exception.message} ${exception.stack}`);
      }
    }

    response.status(statusCode).json({
      code,
      statusCode,
      message,
      data: null,
      errors: null,
    });
  }
}
```

фиг. 3.1.61. - Филтър за обработка на всички останали грешки

3.1.9. Операции на обяви

В останалите точки (без последната, която е за търсене) се разглеждат сравнително подобни функционалности, но със съществени разлики. Тъй като всички контролери в проекта спазват CRUD[lxix] изглеждат и се използват по много подобен начин. Поради това

изходният код ще бъде съкратен и няма да включва всички части. За пълен изходен код може да разгледате приложения към дипломната работа електронен носител, в който се съдържа кода. Може също така да посетите GitHub хранилищата на проекта.

Както всяка функционалност така и обявите, местата и други са в свои собствени модули с бизнес логика, контролери, типове, входни и изходи шаблони (dto) и други.

В контролерът се съдържат следните пътища и операции (фиг. 3.1.62.) – на начален път '/listing' са създадени операции за създаване на обява и взимане на много обяви (под формата на карти), на подпътя, съдържащ ИН (uuid) на обява, '/listing/{uuid}' са създадени операции за извличане на пълната информация на даден пост, актуализиране и изтриване. Освен това има още два пътя – '/listing/{uuid}/rate' и '/listing/{uuid}/save', които се използват съответно за оценяване и запазване на дадена обява.

```
@Controller('listing')
...
export class ListingController {
  constructor(private readonly listingService: ListingService) {}

  @Post()
  ...
  async create(
  ...

  @Get()
  ...
  async findAll(
  ...

  @Get(':uuid')
  ...

  @Post(':uuid/rate')
  ...
  async rate(
  ...

  @Post(':uuid/save')
  ...
  async save(
  ...

  @Patch(':uuid')
  ...
  async update(
  ...

  @Delete(':uuid')
  ...
}
```

фиг. 3.1.62. - Съкратен контролер, показвайки всички пътища

За създаването на обява има път за POST заявка, който е защитен с *@UseGuards(JwtAuthGuard)*, тъй като за да може да бъде създадена дадена обява, потребителят трябва да се е регистрирал. По подразбиране POST заявките връщат код 201. Използвани са декоратори за документация с Open-API Swagger като *@ApiBearerAuth()*, *@ApiBody()* и *@ApiOkResponse()*. Методът за създаване на обява приема тяло от тип Create Listing Dto, като също взима потребителя от заявката, съответно анотирани с *@Body()* и *@Req()*. След което извиква методът create() на бизнес логиката.

```
@Post()
@UseGuards(JwtAuthGuard)
@ApiBearerAuth()
@ApiBody({
  type: CreateListingDto,
})
@ApiOkResponse({
  description: 'Create a new listing',
  type: ResponseListingDto,
  status: 201,
})
async create(
  @Req() req: RequestWithUser,
  @Body() createListingDto: CreateListingDto,
): Promise<ResponseListingDto> {
  return this.listingService.create(req.user, createListingDto);
}
```

фиг. 3.1.63. – Път за създаване на обява

В бизнес логиката (*фиг. 3.1.64.*) се проверява дали категория с такъв ИН съществува и след това трансформира данните в друг тип обект с *plainToClass()*, който е съвместим за качване в базата данни. След това се създава транзакция с Prisma ORM и се взима готовата обява, която бива трансформирана с plainToInstance функцията, за да бъде върната на потребителя.

Транзакциите в Prisma ORM са специални, тъй като те се създават чрез извикването само на един метод за създаване. Следователно създавайки другите таблици посредством връзките си с главната таблица (listing – за обява). Трансформираната информация се предава чисто на

функцията, докато за останалите таблици се или създават връзки с вече съществуващи записи, или се създават нови записи.

```
async create(
  user: User,
  createListingDto: CreateListingDto,
): Promise<ResponseListingDto> {
  const { uuid } = user;
  const { categoryId, tags, images } = createListingDto;

  const category = await this.prisma.category.findUnique({
    where: {
      uuid: categoryId,
    },
  });

  if (!category) {
    throw new BadRequestException('CATEGORY_NOT_FOUND');
  }

  const listingEntity = plainToClass(Listing, createListingDto);

  const createdListing = ...

  return plainToInstance(ResponseListingDto, createdListing);
}
```

фиг. 3.1.64. – Бизнес логика за създаване на обява

Създават се връзки към вече съществуващи потребител и категория, създават се нови записки и връзки за всяка снимка и най-специалната част – таговете, ако даден таг вече съществува – се създава връзка, ако ли не – се създава и нов запис и нова връзка.

```
const createdListing = await this.prisma.listing.create({
      data: {
        ...listingEntity,
        user: {
          connect: {
            uuid,
          },
        },
        category: {
          connect: {
            uuid: categoryId,
          },
        },
        images: {
          create: images.map((image, i) => ({
            url: image,
            alt: image,
            order: i + 1,
          })),
        },
        tags: {
          create: tags.map((tag) => ({
            tag: {
              connectOrCreate: {
                where: {
                  name: tag,
                },
                create: {
                  name: tag,
                },
              },
            },
          })),
        },
      },
      select: ListingSelect,
    });
```

фиг. 3.1.65. – Транзакция за създаване на обява

Извличането на обяви е специално, тъй като то е разделено по страници и лимити. Тук са използвани всички нормални декоратори, но и тези за страници, обяснени в т. 3.1.7. - *@ApiQueryPaginated()* и *@ApiResponsePaginated()*, която приема типа, който се връща – тип за картичка на обява. Методът приема *@Paginate()* – страница и лимит, и *@Query()* – незадължителен параметър за филтриране по категории.

```
@Get()
@ApiQueryPaginated()
@ApiQuery({
  name: 'category',
  required: false,
})
@ApiOkResponsePaginated(ResponseShortListingDto)
async findAll(
  @Paginate() pagination: Pagination,
  @Query('category') category?: string,
): Promise<PaginationResultDto<ResponseShortListingDto>> {
  return this.listingService.findAll(pagination, category);
}
```

фиг. 3.1.66. – Път за извличане на много обяви по страници с лимит

В бизнес логиката за намиране първоначално се взимат ИН на категорията и нейните подкатегории, ако е подаден незадължителния параметър за филтриране. Следователно, в зависимост дали е избрана категория, или не, се определят условията на заявка.

```
async findAll(
  paginate: Pagination,
  category?: string,
): Promise<PaginationResultDto<ResponseShortListingDto>> {
  const categories = category
    ? await this.categoryService.findOneAndSubCategoriesUuids(
        category,
        'LISTING',
      )
    : null;

  const whereClause =
    category && categories
      ? {
          status: Status.PUBLIC,
          deleted_at: null,
          user: {
            deleted_at: null,
          },
          category: { uuid: { in: categories } },
        }
      : commonWhereClause;
...
```

фиг. 3.1.67. – Първа част от бизнес логиката за извличане на много обяви

След това бизнес логиката продължава с извличане на броя на обяви, отговарящи на условията (totalCount) и след това извлича самите обяви, като използва skip и take на Prisma ORM, за да се имплементира извличане по страници.

След това се образува резултат, който взима едната снимка (с order равно на 1) от масива със снимки, като я слага в полето thumbnail. След това този резултат се трансформира с *plainToInstance()* и се връщат метаданни като номер на страница, лимит и общ брой резултати.

```
...

const totalCount = await this.prisma.listing.count({
  where: whereClause,
});

const listings = await this.prisma.listing.findMany({
  select: ShortListingSelect,
  where: whereClause,
  orderBy: {
    created_at: 'desc',
  },
  skip: paginate.page * paginate.limit - paginate.limit,
  take: paginate.limit,
});

const result = listings.map((listing) => {
  const [thumbnail] = listing.images;
  return {
    ...listing,
    thumbnail,
  };
});

return {
  data: plainToInstance(ResponseShortListingDto, result),
  page: paginate.page,
  limit: paginate.limit,
  totalCount,
};
```

фиг. 3.1.68. – Остатък от бизнес логиката за извличане на много обяви

В пътя за извличане на дадена обява се използва NestJS Guard за управление на достъп чрез права за гледане без употребата на *JWT Auth Guard*. Използвани са обикновените декоратори, както и специалните за управление на права - *@Action()* и *@Resource()*, които в този случай

приемат ‘view’ (за гледане) и ‘listing’ (за обява). Единственият Guard, който се използва е този за управление на права – *PermissionsGuard*.

```
@Get(':uuid')
@Action('view')
@Resource('listing')
@UseGuards(PermissionsGuard)
@ApiBearerAuth()
@ApiParam({
  name: 'uuid',
  type: String,
})
@ApiOkResponse({
  description: 'Get a listing by uuid',
  type: ResponseListingDto,
})
async findOne(@Param('uuid') uuid: string): Promise<ResponseListingDto> {
  return this.listingService.findOne(uuid);
}
```

фиг. 3.1.69. – Път за извличане на подробна информация за една обява

Бизнес логиката е сравнително проста, като единствено се извиква допълнителна помощна функция, за извличане на оценката на потребителя, като след това този резултат се трансформира.

```
async findOne(uuid: string): Promise<ResponseListingDto> {
  const listing = await this.prisma.listing.findUniqueOrThrow({
    where: {
      uuid,
      deleted_at: null,
      user: {
        deleted_at: null,
      },
    },
    select: ListingSelect,
  });

  const userRating = await this.userService.getUserRating(listing.user_uuid);

  return plainToInstance(ResponseListingDto, {
    ...listing,
    rating: userRating,
  });
}
```

фиг. 3.1.70. – Бизнес логика за извличане на подробна информация за обява

Оценяването и запазването на обява също са защитени с Permissions Guard, тъй като тези операции могат да бъдат изпълнени само от тези, които имат права да видят дадена публикация. Пътищата не са

показани на фигура, поради тяхната прилика с останалите. В бизнес логиката за оценяване се използва една специална функция на Prisma ORM – *upsert()* – позната като update or insert (ако даден запис съществува – да бъде актуализиран, ако ли не – да бъде създаден).

Тъй като един потребител може да има само една оценка за дадена публикация, се проверява дали тя вече съществува, за да се види дали да се създаде, или актуализира, като след това се връща новата оценка.

```
async rate(
  uuid: string,
  userUuid: string,
  rateListingDto: RateListingDto,
): Promise<RateListingDto> {
  const rating = await this.prisma.listingRating.upsert({
    where: {
      user_uuid_listing_uuid: {
        user_uuid: userUuid,
        listing_uuid: uuid,
      },
    },
    update: {
      rating: rateListingDto.rating,
    },
    create: {
      rating: rateListingDto.rating,
      user: {
        connect: {
          uuid: userUuid,
        },
      },
      listing: {
        connect: {
          uuid,
        },
      },
    },
    select: {
      rating: true,
    },
  });

  return plainToInstance(RateListingDto, rating);
}
```

фиг. 3.1.71. – Бизнес логика за оценяване на обява

Проблем идва в метода за запазване на обява, тъй като предишни стойности не могат да бъдат прочитани чрез функцията *upsert()*. Затова първо се проверява за съществуващ запис – ако такъв съществува, то той

инвертира полето дали е изтрит записът. Ако не съществува се създава нов запис за тази връзка.

```
async save(uuid: string, userUuid: string): Promise<ResponseSavedDto> {
  const existingSave = await this.prisma.userSavedListings.findUnique({
    where: {
      user_uuid_listing_uuid: {
        user_uuid: userUuid,
        listing_uuid: uuid,
      },
    },
  });

  if (existingSave) {
    const saved = await this.prisma.userSavedListings.update({
      where: {
        user_uuid_listing_uuid: {
          user_uuid: userUuid,
          listing_uuid: uuid,
        },
      },
      data: {
        deleted_at: existingSave.deleted_at ? null : new Date(),
      },
      select: {
        deleted_at: true,
      },
    });

    return {
      saved: !saved.deleted_at,
    };
  }

  await this.prisma.userSavedListings.create({
    data: {
      user: {
        connect: {
          uuid: userUuid,
        },
      },
      listing: {
        connect: {
          uuid,
        },
      },
    },
  });

  return {
    saved: true,
  };
}
```

фиг. 3.1.72. – Бизнес логика за запазване на обява

При актуализирането и изтриването на обяви се използват както JWT Auth Guard, така и Permissions Guard за права за достъп. Като при тези

два маршрута са нагласени с анотация *@Action('manage')* за действие на модифициране на дадена обява.

```
@Patch(':uuid')
@Resource('listing')
@Action('manage')
@UseGuards(JwtAuthGuard, PermissionsGuard)
@ApiBearerAuth()
@ApiParam({
  name: 'uuid',
  type: String,
})
@ApiBody({
  type: UpdateListingDto,
})
@ApiOkResponse({
  description: 'Update a listing by uuid',
  type: ResponseListingDto,
  status: 200,
})
async update(
  @Param('uuid') uuid: string,
  @Req() req: RequestWithUser,
  @Body() updateListingDto: UpdateListingDto,
): Promise<ResponseListingDto> {
  return this.listingService.update(uuid, req.user.uuid, updateListingDto);
}

@Delete(':uuid')
@Resource('listing')
@Action('manage')
@UseGuards(JwtAuthGuard, PermissionsGuard)
@ApiBearerAuth()
@ApiParam({
  name: 'uuid',
  type: String,
})
@ApiOkResponse({
  description: 'Delete a listing by uuid',
  status: 204,
})
async remove(@Param('uuid') uuid: string) {
  return this.listingService.remove(uuid);
}
```

фиг. 3.1.73. – Маршрути за актуализиране и изтриване на обява

При изтриването на обява, се извършва т.нар. „меко изтриване“, където самият запис не се заличава от базата данни, а му се променят две полета – статус на изтрит, и изтрит на – текущо време.

```
async remove(uuid: string) {
  return this.prisma.listing.update({
    where: {
      uuid,
    },
    data: {
      status: 'DELETED',
      deleted_at: new Date(),
    },
  });
}
```

фиг. 3.1.74. – Бизнес логика за изтриване на обява

Методът за актуализиране е почти еднакъв с този на създаване, с единствената разлика, че се извиква функцията update от Prisma ORM. Методът за създаване може да бъде разгледан на *фиг. 3.1.64.* и *3.1.65.*

3.1.10. Операции на места

Операциите, които могат да се извършват на места много наподобяват тези, разгледани в т. 3.1.9. за обяви, поради което в тази точка ще бъдат разгледани по-специфичните случаи и заявки, които се изпълняват за места, като нормалните CRUD функционалности няма да бъдат разгледани.

```
@Controller('place')
// ...
export class PlaceController {
  constructor(private readonly placeService: PlaceService) {}

  @Post()
  // ...
  async create( /* ... */ ) {}
  /// ...

  @Get()
  // ...
  async findAll( /* ... */ ) {}
  // ...

  @Get(':uuid')
  // ...
  async findOne(@Param('uuid') uuid: string) {}
  // ...

  @Post(':uuid/save')
  // ...
  async save( /* ... */ ) {}
  // ...

  @Post(':uuid/here')
  // ...
  async here( /* ... */ ) {}
  // ...

  @Get(':uuid/here')
  // ...
  async getHere( /* ... */ ) {}
  // ...

  @Patch(':uuid')
  // ...
  async update( /* ... */ ) {}
  // ...

  @Delete(':uuid')
  // ...
  remove(@Param('uuid') uuid: string) {}
  // ...
}
```

фиг. 3.1.75. – Контролер за места

В контролера се съдържат нормалните CRUD операции – създаване, взимане (на една или повече обяви), актуализирането на местата, както и тяхното изтриване. Специални операции, които може да се извършват върху местата са: запазване и отбелязване на присъственост както и заявка за взимане на всички, които са били на това място.

Главна промяна, която трябва да бъде отбелязана е, че CRUD операциите се изпълняват за места, а не за обяви, както е описано в т. 3.1.9. и съществуват много разлики в данните, които се получават и изпращат. Те могат да бъдат разгледани във файловата структура на модула за места. Една от забележките за CRUD функционалността е, че при извличането на много места има два типа, данни, които могат да бъдат върнати – информация за картички за места или информация за пинове на карта.

```
...
const places = await this.prisma.place.findMany({
  select: format === 'short' ? ShortPlaceSelect : CardPlaceSelect,
  where: whereClause,
  orderBy: {
    created_at: 'desc',
  },
  skip: pagination.page * pagination.limit - pagination.limit,
  take: pagination.limit,
});

if (format === 'short') {
  return {
    data: plainToInstance(ResponseShortPlaceDto, places),
    ...pagination,
    totalCount,
  };
}
...
```

фиг. 3.1.76. – Връщане на различен тип данни за много места спрямо типа на извличане – за карта или за картички

Други модификации, които са направени спрямо обикновения CRUD, за извличането и изчисляването на средноаритметична оценка за всяко място, което се извлича като картичка. Където с метода *groupBy()* на Prisma ORM се групират оценките на местата по идентификационен номер, като след това за всяко едно от тях се изчислява средна оценка от

SQL AVG функцията. След това, заедно с главните снимки за изглед (thumbnail), се поставят оценките за всяко едно място в неговия обект и се връщат.

```
...
const ratings = await this.prisma.placeReview.groupBy({
  by: ['placeUuid'],
  _avg: {
    rating: true,
  },
  where: {
    placeUuid: { in: places.map((place) => place.uuid) },
  },
});

const ratingsMap = new Map(
  ratings.map((r) => [r.placeUuid, r._avg.rating]),
);

const result = places.map((place) => {
  const averageRating = ratingsMap.get(place.uuid) || 0;
  const [thumbnail] = place.images;

  return {
    ...place,
    thumbnail,
    rating: roundRating(averageRating),
  };
});

return {
  data: plainToInstance(ResponseCardPlaceDto, result),
  ...pagination,
  totalCount,
};
```

фиг. 3.1.77. – Извличане и изчисляване на средна оценка за всяка картичка на място

Подобно на извличането на една обява, така и при извличането на едно място, то се прилага заедно с него потребителя, създал даденото съдържание. Освен това към мястото се добавят както собственик, ако има такъв, така и последните 3 ревюта, последните 3 посетители. Което означава, че за всички тези компоненти трябва да се извлече и изчисли тяхната оценка (за потребителите), както и средната оценка на дадено място. Средната оценка на мястото се изчислява с агрегатния метод на Prisma ORM за дадено място по ИН, докато при потребителите – създател,

собственик и последните 3 посетители, се извиква метод от бизнес логиката на потребителите за изчисляване на тяхната средноаритметична оценка – *getUserRating()*, ато след това те се добавят към отговора на заявката.

```
// ...
const averageRating = await this.prisma.placeReview.aggregate({
  where: {
    placeUuid: uuid,
  },
  _avg: {
    rating: true,
  },
});

// get creator user rating, owner user rating and visitors user rating
const creatorRating = await this.userService.getUserRating(
  place.creator.uuid,
);
const ownerRating = await this.userService.getUserRating(place.owner?.uuid);

const visitorsRatingsPromise = place.visitors.map(async (visitor) => {
  const newVisitor = plainToInstance(VisitorDto, visitor);
  return this.userService.getUserRating(newVisitor.user.uuid);
});
const visitorRatings = await Promise.all(visitorsRatingsPromise);

return plainToClass(ResponsePlaceDto, {
  ...place,
  creator: {
    ...place.creator,
    rating: roundRating(creatorRating),
  },
  owner: {
    ...place.owner,
    rating: roundRating(ownerRating),
  },
  visitors: place.visitors.map((visitor, i) => ({
    ...visitor,
    rating: roundRating(visitorRatings[i]),
  })),
  rating: roundRating(averageRating._avg.rating),
});
```

фиг. 3.1.78. - Извличане и изчисляване на средни оценки за място, създател, собственик и посетители на място

За метода на отбелязване на даден потребител за неговото присъствие, се взима ИН на потребител от JWT токен за авторизация, като се взима и ИН на дадено място от URL параметър. Методът за отбелязване наподобява на смесица между метода за запазване на място / обява и оценяване на обява. Първоначално се проверява дали вече съществува

такава много-към-много връзка, като ако съществува, то полето за меко изтриване на връзката се инвертира, а ако не съществува, то тя се създава. За тази цел се използва функцията на Prisma ORM – *upsert()*.

```
async here(uuid: string, userUuid: string): Promise<ResponseHereDto> {
  const existingHere = await this.prisma.placeVisitor.findUnique({
    where: {
      placeUuid_userUuid: {
        placeUuid: uuid,
        userUuid,
      },
    },
  });

  const res = await this.prisma.placeVisitor.upsert({
    where: {
      placeUuid_userUuid: {
        placeUuid: uuid,
        userUuid,
      },
    },
    create: {
      user: {
        connect: {
          uuid: userUuid,
        },
      },
      place: {
        connect: {
          uuid,
        },
      },
    },
    update: {
      deleted_at: existingHere?.deleted_at ? null : new Date(),
    },
    select: {
      deleted_at: true,
    },
  });

  return {
    here: !res.deleted_at,
  };
}
```

фиг. 3.1.79. – Отбелязване на даден обект, че е посетен от потребителя

Другият метод, който е уникален за контролера и бизнес логиката на местата, е този за извличане на всички потребители, които са се отбелязали като „посетили“ дадено място. Методът също е направен за връщане на данни, разделени на страници и с лимит на бройка. След което се извличат всички потребители, които са били на дадено място по

ИН, които не са изтрити, за дадена страница с даден лимит, като след това се изчисляват средните оценки за всеки потребител и се връща форматиран резултат.

```
async getHere(
  pagination: Pagination,
  uuid: string,
): Promise<PaginationResultDto<ResponseCardUserDto>> {
  const totalCount = await this.prisma.placeVisitor.count({
    where: {
      placeUuid: uuid,
      deleted_at: null,
    },
  });

  const here = await this.prisma.placeVisitor.findMany({
    select: { /* ... */ },
    where: {
      placeUuid: uuid,
      deleted_at: null,
    },
    orderBy: {
      created_at: 'desc',
    },
    skip: pagination.page * pagination.limit - pagination.limit,
    take: pagination.limit,
  });

// ... calc user ratings ...

return {
  data: plainToInstance(ResponseCardUserDto, result),
  ...pagination,
  totalCount,
};
```

фиг. 3.1.80. – Извличане на потребителите, посещавали дадено място

Останалите заявки може да бъдат разгледани на електронния носител или в GitHub хранилището на проекта, като те наподобяват на тези от т. 3.1.9.

3.1.11. Операции на коментари и ревюта

Подобно на услугите и контролерите за обяви и места, тези за коментари и ревюта си наподобяват, като ще бъде разгледана само първата услуга и контролер – тази за коментари.

В контролера за коментари / ревюта се съдържат само обикновени CRUD операции – за извличане на един или много коментари / ревюта (спрямо дадена обява / място), за публикуване на коментар / ревю на дадена обява / място, за актуализиране и изтриване на даден коментар / място.

```
@Controller('listing/:uuid/comment')
// ...
export class CommentController {
  constructor(private readonly commentService: CommentService) {}

  @Post()
  // ...
  async create ( /* ... */) { }
  // ...

  @Get()
  // ...
  async findAll ( /* ... */) { }
  // ...

  @Get(':sub_uuid')
  // ...
  async findOne ( /* ... */) { }
  // ...

  @Patch(':sub_uuid')
  // ...
  async update ( /* ... */) { }
  // ...

  @Delete(':sub_uuid')
  // ...
  async remove ( /* ... */) { }
  // ...
}
```

фиг. 3.1.81. – Контролер за коментари

Както е обяснено в т. 3.1.9. на всеки път са сложени различни декоратори както за документация със Swagger / Open-API, така и за различни защити – примерно с *@UseGuards()*. Обаче за създаване на коментар се добавя и защита, която проверява дали даден пост, може да бъде разглеждан от потребителя, преди да му се даде възможност да коментира. Това може да бъде забелязано при всяка добавяща или модифицираща операция в контролерите за коментари и ревюта. Те са имплементирани с *PermissionsGuard* и с анотациите: *@Action()* и

@Resource(). Пътят за създаване на коментар приема от URL параметрите – ИН на обява, а от тялото на заявката – съдържанието на дадения коментар. Потребителският ИН се извлича от *JwtAuthGuard*.

```
@Post()
@Action('view')
@Resource('listing')
@UseGuards(JwtAuthGuard, PermissionsGuard)
@ApiBearerAuth()
@ApiBody({
  type: CreateCommentDto,
})
@ApiResponse({
  status: 201,
  type: ResponseCommentDto,
})
async create(
  @Req() req: RequestWithUser,
  @Body() createCommentDto: CreateCommentDto,
  @Param('uuid') uuid: string,
): Promise<ResponseCommentDto> {
  return this.commentService.create(req.user.uuid, uuid, createCommentDto);
}
```

фиг. 3.1.82. – Път за създаване на коментар

За създаване на коментара се използва функцията create() в Prisma ORM, като се подава съдържанието и се правят връзки с обявата и потребителя, създал дадения коментар. След това се връща резултат.

```
async create(
  userUuid: string,
  listingUuid: string,
  createCommentDto: CreateCommentDto,
): Promise<ResponseCommentDto> {
  const createdComment = await this.prisma.listingComment.create({
    data: {
      content: createCommentDto.content,
      user: {
        connect: {
          uuid: userUuid,
        },
      },
      listing: {
        connect: {
          uuid: listingUuid,
        },
      },
    },
    select: CommentSelect,
  });

  return plainToClass(ResponseCommentDto, createdComment);
}
```

фиг. 8.1.83. – Бизнес логика за създаване на коментар

За извличането на всички коментари на дадена обява, потребителят трябва да има достъп до преглеждане на обявата. Имплементирана е по същия начин както при създаването на коментар. Пътят връща всички коментари, разделени по страници, спрямо дадения лимит (Pagination).

```
@Get()
@Action('view')
@Resource('listing')
@UseGuards(PermissionsGuard)
@ApiBearerAuth()
@ApiQueryPaginated()
@ApiOkResponsePaginated(ResponseCommentDto)
async findAll(
  @Param('uuid') uuid: string,
  @Paginate() pagination: Pagination,
): Promise<PaginationResultDto<ResponseCommentDto>> {
  return this.commentService.findAll(uuid, pagination);
}
```

фиг. 3.1.84. – Път за извличане на всички коментари на дадена обява

Методът за извличане на всички коментари за дадена обява следва принципа на всички paginated[lxx] отговори.

```
async findAll(
  uuid: string,
  pagination: Pagination,
): Promise<PaginationResultDto<ResponseCommentDto>> {
  const whereClause = {
    deleted_at: null,
    status: Status.PUBLIC,
    listing: {
      uuid,
    },
  };

  const totalCount = await this.prisma.listingComment.count({
    where: whereClause,
  });

  const comments = await this.prisma.listingComment.findMany({
    select: CommentSelect,
    where: whereClause,
    orderBy: {
      created_at: 'desc',
    },
    skip: pagination.page * pagination.limit - pagination.limit,
    take: pagination.limit,
  });

  return {
    data: plainToInstance(ResponseCommentDto, comments),
    ...pagination,
    totalCount,
  };
}
```

фиг. 3.1.85. – Бизнес логика за извличане на всички коментари по страница

Пътят за извличане на даден коментар се защитава с достъпност до самия коментар – с *@Resource('comment')*. При възможност за извличането на коментара, се връща неговото съдържание.

```
@Get(':sub_uuid')
@Action('view')
@Resource('comment')
@UseGuards(PermissionsGuard)
@ApiBearerAuth()
@ApiParam({
  name: 'sub_uuid',
  required: true,
  description: 'Comment UUID',
})
@ApiResponse({
  status: 200,
  type: ResponseCommentDto,
})
async findOne(@Param('sub_uuid') uuid: string):
Promise<ResponseCommentDto> {
  return this.commentService.findOne(uuid);
}
```

фиг. 3.1.86. – Път за извличане на даден коментар

Методът за извличане на даден коментар по неговия ИН е отново прост CRUD метод, който е разглеждан в т. 3.1.9., където се избира коментар, който не е изтрит и му пасва идентификационния номер.

```
async findOne(uuid: string): Promise<ResponseCommentDto> {
  const comment = await this.prisma.listingComment.findUniqueOrThrow({
    where: {
      uuid,
      status: Status.PUBLIC,
      deleted_at: null,
    },
  });

  return plainToClass(ResponseCommentDto, comment);
}
```

фиг. 3.1.87. – Бизнес логика за извличане на даден коментар

В пътя за актуализиране на даден коментар се приемат както ИН на дадена обява, така и ИН на дадения коментар, който ще бъде актуализиран. Пътят е защитен както с *JwtAuthGuard*, тъй като потребител задължително трябва да е в профил, така и с *PermissionsGuard*, който се

използва за извършване на проверка дали даденият потребител има права за модифициране на дадения коментар – неговото съдържание и статус (дали е публичен).

```
@Patch(':sub_uuid')
@Action('manage')
@Resource('comment')
@UseGuards(JwtAuthGuard, PermissionsGuard)
@ApiBearerAuth()
@ApiParam({
  name: 'sub_uuid',
  required: true,
  description: 'Comment UUID',
})
@ApiBody({
  type: UpdateCommentDto,
})
@ApiResponse({
  status: 200,
  type: ResponseCommentDto,
})
async update(
  @Param('sub_uuid') uuid: string,
  @Body() updateCommentDto: UpdateCommentDto,
): Promise<ResponseCommentDto> {
  return this.commentService.update(uuid, updateCommentDto);
}
```

фиг. 3.1.88. – Път за актуализиране на определен коментар

В бизнес логиката може да се актуализират съдържанието и статуса (дали е публичен) на коментара, чийто идентификационен номер съвпада с подадения. След успешно изпълнение се връща нов отговор под формата на извлечен коментар.

```
async update(uuid: string, updateCommentDto: UpdateCommentDto) {
  const updatedComment = await this.prisma.listingComment.update({
    where: {
      uuid,
    },
    data: {
      content: updateCommentDto.content,
      status: updateCommentDto.status ?? undefined,
    },
    select: CommentSelect,
  });

  return plainToClass(ResponseCommentDto, updatedComment);
}
```

фиг. 3.1.89. – Бизнес логика за актуализиране на определен коментар

Пътят за изтриване на даден коментар е също защитен както с JWT токен, така и с условия за права на модифициране на коментар. В бизнес логиката, която се извиква от контролера, се извършва т.нар. „меко“ изтриване, където се попълва с текущата дата и час полето „deleted_at” в таблицата за коментари.

```
@Delete(':sub_uuid')
@Action('manage')
@Resource('comment')
@UseGuards(JwtAuthGuard, PermissionsGuard)
@ApiBearerAuth()
@ApiParam({
  name: 'sub_uuid',
  required: true,
  description: 'Comment UUID',
})
@ApiResponse({
  status: 200,
})
async remove(@Param('sub_uuid') uuid: string) {
  return this.commentService.remove(uuid);
}
```

фиг. 3.1.90. – Път за изтриване на определен коментар

3.1.12. Операции на потребители

Контролерът за потребители е един от най-големите и пъстри контролери в този приложно-програмен интерфейс. Логически може да се раздели на две части – заявки за личен (регистриран) профил, както и за публичен такъв, като по него се работи с подаване на потребителско име.

Пътищата, които са за личен профил са защитени с *@UseGuards(JwtAuthGuard)*. Като с този NestJS Guard се извлича и идентификационния номер на потребителя, на когото съответно се извършва дадената заявка. Пътищата са следните:

- *<u>/user/me</u>* – заявка за извличане на данни за текущо регистрирания потребител

- /user/me/listing – заявка за извличане на обявите, публикувани от текущо регистрирания потребител
- /user/me/place – заявка за извличане на местата, публикувани от текущо регистрирания потребител
- /user/me/saved/listing – заявка за извличане на запазените обяви от текущо регистрирания потребител
- /user/me/saved/place – заявка за извличане на запазените места от текущо регистрирания потребител

```
@Controller('user')
// ...
export class UserController {
  // ...

  @Get('me')
  @UseGuards(JwtAuthGuard)
  // ...
  async getMe ( /* ... */ ) { }
  // ...

  @Get('me/listing')
  @UseGuards(JwtAuthGuard)
  // ...
  async getMyListings ( /* ... */ ) { }
  // ...

  @Get('me/place')
  @UseGuards(JwtAuthGuard)
  // ...
  async getMyPlaces ( /* ... */ ) { }
  // ...

  @Get('me/saved/listing')
  @UseGuards(JwtAuthGuard)
  // ...
  async getSavedListings ( /* ... */ ) { }
  // ...

  @Get('me/saved/place')
  @UseGuards(JwtAuthGuard)
  // ...
  async getSavedPlaces ( /* ... */ ) { }
  // ...

 // ...
```

фиг. 3.1.91. – Част от контролера за потребител, с показани права на достъп – пътища за личен профил

За извличането на профила на регистрирания потребител се изпълнява проста заявка с Prisma ORM – findUniqueOrThrow(), където се извлича само по идентификационния номер на потребителя от JWT

токена за достъп. При несъществуващ такъв потребител – връща се грешка 404. При успешно извличане се изчислява и оценка с помощния метод *getUserRating()*.

```
async getUserProfile(uuid: string): Promise<ResponseProfileDto> {
  const user = await this.prisma.user.findUniqueOrThrow({
    where: {
      uuid,
    },
  });

  const userRating = await this.getUserRating(uuid);

  return plainToInstance(ResponseProfileDto, {
    ...user,
    rating: userRating,
  });
}
```

фиг. 3.1.92. – Бизнес логика за извличане на текущо регистрирания потребител

В помощният метод за извличане на средноаритметичната оценка на потребител се използва функцията на Prisma ORM за групиране на данните по определен критерий, като след това се изчислява средноаритметичната оценка на този потребител спрямо неговия ИН и оценените потребители.

```
async getUserRating(uuid: string | undefined): Promise<number> {
  if (!uuid) return 0;

  const userRating = await this.prisma.userRating.groupBy({
    by: ['user_rated_uuid'],
    _avg: {
      rating: true,
    },
    where: {
      user_rated_uuid: uuid,
    },
  });

  console.log('user ratings', userRating);

  return userRating[0] ? roundRating(userRating[0]._avg.rating) : 0;
}
```

фиг. 3.1.93. – Помощен метод за извличане на средноаритметичната оценка на даден потребител (по идентификационния му номер)

Пътищата за извличане на обяви и места, създадени от регистрирания потребител, използват вече направените методи от бизнес логиката за услугите на места и обяви тип „findAll", които приемат незадължителен параметър – потребителско име. От JwtAuthGuard се извлича потребителското име на клиента, което се подава на метода, който извлича или обявите, или местата.

Пътищата, които се използват за операции на публични потребителски профили (един или много), са следните:

- */user/profile* – извличане на всички потребителски картички (само публична профилна информация) по страница с лимит
- */user/{username}* – извличане на публичен потребителски профил по подадено потребителско име като URL параметър
- */user/{username}/listing* – извличане на всички публични обяви (по страници), създадени от публичен потребителски профил, по подадено потребителско име като URL параметър
- */user/{username}/place* – извличане на всички публични места (по страници), създадени от публичен потребителски профил, по подадено потребителско име като URL параметър
- */* – извличане на цялата потребителска информация – публична и частна. Този път е защитен за използване само от потребители с администраторска роля. Използван е специалния (custom) декоратор *@Roles('ADMIN')*, в който е подадена само администраторската роля, освен това е сложен и *@UseGuards(JwtAuthGuard, RoleGuard)* за авторизация.
- *[POST] /user/rate/{username}* – защитен път за създаване на нова оценка или актуализиране на такава за друг потребител

- *[GET] /user/rate/{username}* – защитен път за извличане на оценка, направена от потребител (взет от JwtAuthGuard), за подаден от URL параметър потребител (по потребителско име)

```
@Get('profile')
// ...
async getProfiles ( /* ... */ ) { }
// ...

@Get(':username')
// ...
async getProfile(
// ...

@Get(':username/listing')
// ...
async getProfileListings ( /* ... */ ) { }
// ...

@Get(':username/place')
// ...
async getProfilePlaces ( /* ... */ ) { }
// ...

@Get()
@UseGuards(JwtAuthGuard, RoleGuard)
@Roles('ADMIN')
// ...
async getUsers ( /* ... */ ) { }
// ...

@Post('rate/:username')
@UseGuards(JwtAuthGuard)
// ...
async rate ( /* ... */ ) { }
// ...

@Get('rate/:username')
@UseGuards(JwtAuthGuard)
// ...
async getRate ( /* ... */ ) { }
// ...
```

фиг. 3.1.95. – Част от контролера за потребител, с показани права на достъп – пътища за личен профил

За извличане на публичен потребителски профил се използва помощната функция за намиране на потребител по потребителско име, след което се проверява дали той е празен, ако да – такъв потребител не съществува и се връща грешка. При успешно извличане на потребител се

изчислява неговата средноаритметична оценка с помощния метод – *getUserRating()*, след което се връща резултат.

```
async getPublicProfile(username: string): Promise<ResponsePublicProfileDto> {
  const user = await this.findByUsername(username);

  if (!user) {
    throw new NotFoundException('USER_NOT_FOUND');
  }

  const userRating = await this.getUserRating(user.uuid);

  return plainToClass(ResponsePublicProfileDto, {
    ...user,
    rating: userRating,
  });
}
```

фиг. 3.1.96. – Бизнес логика за извличане на публичен профил по потребителско име

За извличането на много потребителски картички по страници се изпълнява обичайната paginated процедура за извличане чрез Prisma ORM и *findMany()*, но в този случай за потребители. След това се изчислява оценката на всеки потребител и се връща резултат.

```
async getPublicProfiles(
  pagination: Pagination,
): Promise<PaginationResultDto<ResponsePublicProfileDto>> {
  const whereClause = {
    deleted_at: null,
  };

  const totalCount = await this.prisma.user.count({
    where: whereClause,
  });

  const users = await this.prisma.user.findMany({
    where: whereClause,
    orderBy: {
      created_at: 'desc',
    },
    skip: pagination.page * pagination.limit - pagination.limit,
    take: pagination.limit,
  });

  // ...
  // calculate and put user ratings for each user
  // ...

  return {
    data: plainToInstance(ResponsePublicProfileDto, result),
    ...pagination,
    totalCount,
  };
}
```

фиг. 3.1.97. – Бизнес логика за извличане на потребителски картички

Всички останали функции, подобно на личния профил, използват вече създадените методи за извличане на всички постове (обяви или места), като им се подава аргумент – потребителско име и публична видимост.

Методите за оценяване са подобни и почти еднакви с тези за оценяване на пост (обява или място) от потребител. Но с две допълнителни проверки – проверка за съществуващ потребител, който подлежи на оценка. Както и проверка дали потребителят се опитва да оцени себе си, което не е позволено.

```
const targetUser = await this.findByUsername(targetUsername);

if (!targetUser) {
  throw new NotFoundException('USER_NOT_FOUND');
}

if (srcUuid === targetUser.uuid) {
  throw new ConflictException('CANNOT_RATE_SELF');
}
```

фиг. 3.1.98. – Допълнителни проверки върху метод за оценяване

3.1.13. Търсене в уеб приложението

Търсенето в уеб приложението е разделено на два вида – едно глобално и друго – по категория (обява, място или потребител). Идеята на глобалното търсене е да покаже малък на брой резултати от всяка категория на потребителя, за да може след това той да прецени къде да търси по-подробно – т.е. да избере търсене по категория.

В контролера за търсене съществува само един път – този за търсене, който се изпълнява с GET заявка. При него е задължително да бъде обособена търсещата дума или думи като част от URL на заявката. Като опциите за страница и лимит (Pagination) се използват само при подадена категория. В този път се избира и коя бизнес логика трябва да

поеме изпълнението на търсенето, спрямо това коя категория е подадена, или дали въобще е подадена.

```
@Controller('search')
// ...
export class SearchController {
  constructor(private readonly searchService: SearchService) {}

  @Get()
  // ...
  async search(
    @Query('search') search: string,
    @Paginate() pagination: Pagination,
    @Query('category') category?: 'Freelancers' | 'Equipment' | 'Places',
  ): Promise<
    | PaginationResultDto<
        ResponseShortListingDto | ResponseCardPlaceDto | ResponseCardUserDto
      >
    | SearchAllDto
  > {
    search = search.split(' ').join(' & ');

    switch (category) {
      case 'Freelancers':
        return this.searchService.searchUsers(search, pagination);
      case 'Equipment':
        return this.searchService.searchListings(search, pagination);
      case 'Places':
        return this.searchService.searchPlaces(search, pagination);
      default:
        return this.searchService.searchAll(search);
    }
  }
}
```

фиг. 3.1.99. – Контролер за търсене в уеб приложението

При неподаване на категория, се извиква метода за глобално търсене в уеб приложението. В него се извикват по-малките методи за търсене по-категория, с предварително зададени лимити и номер на страница. След тяхното изпълнение се връщат лимитирани резултати от всяка категория.

```
async searchAll(query: string): Promise<SearchAllDto> {
  const [listings, places, users] = await Promise.all([
    this.searchListings(query, {
      page: 1,
      limit: 6,
    }),
    this.searchPlaces(query, {
      page: 1,
      limit: 4,
    }),
    this.searchUsers(query, {
      page: 1,
      limit: 6,
    }),
  ]);

  return {
    listings,
    places,
    users,
  };
}
```

фиг. 3.1.100. – Бизнес логика за глобално търсене

За търсене, на която и да е категория, се използва нормалната функция за извличане на много данни на Prisma ORM – *findMany()*, но с една съществена разлика – всички елементи се сортират по релевантност (най-подходящ резултат) с вградената функция на PostgreSQL или MySQL за Full Text Search[lxxi], като се поставя начина на сортиране, полетата, които да се използват за сравнение и с какъв текст да се сравнява (този, подаден от потребителя).

За търсенето на обяви се прилага познатият метод за paginated извличане с пропускане на N елементи и взимане на M елементи. Както е описано в горния параграф, във функцията за извличане се използва сортиране по релевантност, като зададените полета за търсене са: заглавието и описанието на обявите. След тяхното извличане от базата данни, се оправя главната снимка и се връща готов резултат на потребителя.

```
async searchListings(
  query: string,
  pagination: Pagination,
): Promise<PaginationResultDto<ResponseShortListingDto>> {
  const totalCount = await this.prisma.listing.count({
    where: commonWhereClause,
  });

  const foundListings = await this.prisma.listing.findMany({
    select: ListingSelect,
    where: commonWhereClause,
    orderBy: {
      _relevance: {
        fields: ['title', 'description'],
        search: query,
        sort: 'desc',
      },
    },
    skip: pagination.page * pagination.limit - pagination.limit,
    take: pagination.limit,
  });

  const result = foundListings.map((listing) => {
    const [thumbnail] = listing.images;
    return {
      ...listing,
      thumbnail,
    };
  });

  return {
    data: plainToInstance(ResponseShortListingDto, result),
    ...pagination,
    totalCount,
  };
}
```

фиг. 3.1.101. – Метод за търсене на обяви

На подобен принцип са изработени и останалите методи за търсене на съдържание, като търсенето на място отново е paginated. Резултатите се сортират по релевантност спрямо полетата за наименование и описание на местата. Освен това се оправят както главната снимка, която се използва за картичката (thumbnail), така и оценката на мястото.

```
async searchPlaces(
  query: string,
  pagination: Pagination,
): Promise<PaginationResultDto<ResponseCardPlaceDto>> {
  const whereClause = {
    deleted_at: null,
    status: Status.PUBLIC,
    creator: {
      deleted_at: null,
    },
  };

  const totalCount = await this.prisma.place.count({
    where: whereClause,
  });

  const foundPlaces = await this.prisma.place.findMany({
    where: whereClause,
    select: CardPlaceSelect,
    orderBy: {
      _relevance: {
        fields: ['name', 'description'],
        search: query,
        sort: 'desc',
      },
    },
    skip: pagination.page * pagination.limit - pagination.limit,
    take: pagination.limit,
  });

  // rating and thumbnail map
  // ...
  // end rating and thumbnail map

  return {
    data: plainToInstance(ResponseCardPlaceDto, result),
    ...pagination,
    totalCount,
  };
}
```

фиг. 3.1.102. – Метод за търсене на места

В метода за потребители се сортира по релевантност потребителското име, името на потребителя и неговото описание.

```
async searchUsers(
  query: string,
  pagination: Pagination,
): Promise<PaginationResultDto<ResponseCardUserDto>> {
  const totalCount = await this.prisma.user.count({
    where: {
      deleted_at: null,
    },
  });

  const foundUsers = await this.prisma.user.findMany({
    where: {
      deleted_at: null,
    },
    orderBy: {
      _relevance: {
        fields: ['username', 'name', 'bio'],
        search: query,
        sort: 'desc',
      },
    },
    skip: pagination.page * pagination.limit - pagination.limit,
    take: pagination.limit,
  });

  return {
    data: plainToInstance(ResponseCardUserDto, foundUsers),
    ...pagination,
    totalCount,
  };
}
```

фиг. 3.1.103. – Метод за търсене на потребители

3.1.14. Обобщение на всички контролери

user

GET	/api/user/me
GET	/api/user/me/listing
GET	/api/user/me/place
GET	/api/user/me/saved/listing
GET	/api/user/me/saved/place
GET	/api/user/profile
GET	/api/user/{username}
GET	/api/user/{username}/listing
GET	/api/user/{username}/place
GET	/api/user
POST	/api/user/rate/{username}
GET	/api/user/rate/{username}

admin

GET	/api/user
POST	/api/category/{type}
PATCH	/api/category/{type}/{uuid}
DELETE	/api/category/{type}/{uuid}
POST	/api/service
PATCH	/api/service/{uuid}
DELETE	/api/service/{uuid}

auth

POST	/api/auth/login
POST	/api/auth/validate/account
POST	/api/auth/validate/personal
POST	/api/auth/signup
POST	/api/auth/logout
GET	/api/auth/refresh
GET	/api/auth/blacklist

listing

- POST /api/listing
- GET /api/listing
- GET /api/listing/{uuid}
- PATCH /api/listing/{uuid}
- DELETE /api/listing/{uuid}
- POST /api/listing/{uuid}/rate
- POST /api/listing/{uuid}/save

listing/comment

- POST /api/listing/{uuid}/comment
- GET /api/listing/{uuid}/comment
- GET /api/listing/{uuid}/comment/{sub_uuid}
- PATCH /api/listing/{uuid}/comment/{sub_uuid}
- DELETE /api/listing/{uuid}/comment/{sub_uuid}

category

- POST /api/category/{type}
- GET /api/category/{type}
- GET /api/category/{type}/{uuid}
- PATCH /api/category/{type}/{uuid}
- DELETE /api/category/{type}/{uuid}

place

- POST /api/place
- GET /api/place
- GET /api/place/{uuid}
- PATCH /api/place/{uuid}
- DELETE /api/place/{uuid}
- POST /api/place/{uuid}/save
- POST /api/place/{uuid}/here
- GET /api/place/{uuid}/here

place/review

- POST /api/place/{uuid}/review
- GET /api/place/{uuid}/review
- GET /api/place/{uuid}/review/my-review
- GET /api/place/{uuid}/review/{sub_uuid}
- PATCH /api/place/{uuid}/review/{sub_uuid}
- DELETE /api/place/{uuid}/review/{sub_uuid}

service

POST /api/service

GET /api/service

PATCH /api/service/{uuid}

DELETE /api/service/{uuid}

search

GET /api/search

file

GET /api/file/upload

3.2. Клиентска част

3.2.1. Структура на файловата система

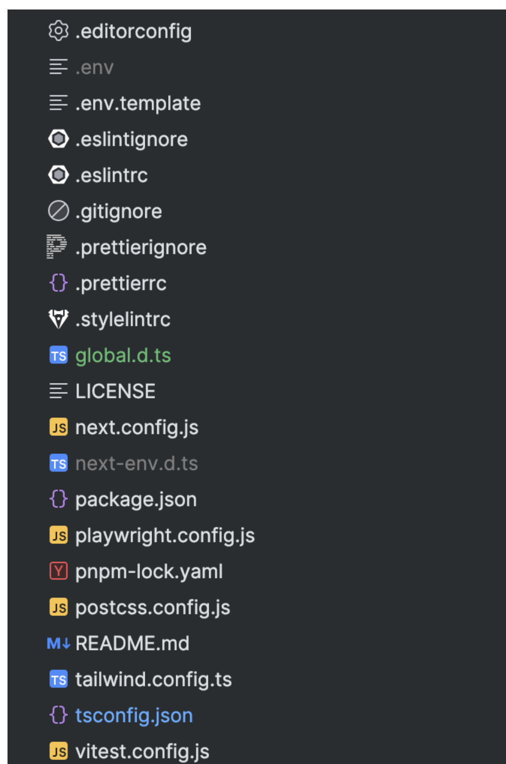
По подобие на сървърния софтуер, този проект за потребителския интерфейс също е пълен с разнообразни конфигурационни файлове.

Тези, които са разгледани в т. 3.1.1. са: “.env”, “.eslintignore”, “.prettierignore”, “.gitignore”, “.prettierrc”, “.eslintrc”, “package.json”, “pnpm-lock.yaml” и “tsconfig.json”.

Новите конфигурационни файлове са следните:

- *.editorconfig* – конфигурационен стандарт за настройки на развойните среди за разработка
- *.env.template* – примерен публичен “.env” файл, който обозначава как трябва да бъде форматиран и какво да съдържа един “.env” конфигурационен файл
- *.stylelintrc* - конфигурация за статичен на стилови CSS файлове и разширения
- *global.d.ts* – конфигурационен файл за деклариране на глобални типове в TypeScript
- *next.config.js* - конфигурационен файл за NextJS
- *playwright.config.js* – конфигурационен файл за Playwright – за автоматизирано тестване в уеб браузъри за визуално тестване.
- *postcss.config.js* – конфигурационен файл за PostCSS, който се използва от TailwindCSS
- *tailwind.config.ts* – конфигурационен файл за CSS софтуерната рамка - TailwindCSS

- *vitest.config.js* – конфигурационен файл за Vitest, използван за тестване на проекта.



фиг. 3.2.1. – Конфигурационни файлове във файловата структура на проекта

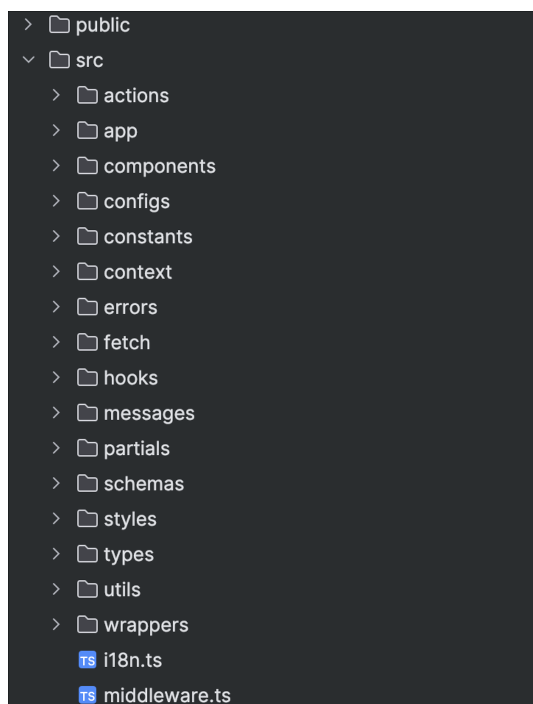
В проекта има две главни директории – *public* и *src*, където съответно се съдържа публичната информация, като статични снимки, файлове и текстове, и изходния TypeScript код.

"src" директория е разделена на много подкатегории, но съдържа и два файла – „i18n.ts" (конфигурационен файл за интернационализация на уеб приложението с next-intl) и "middleware.ts" (софтуер, който се изпълнява между заявките, като „човек по средата" или „междинен човек", подобно на Интерсепторите в NestJS. Директориите в "src" са следните:

- *actions* – съдържат се всички сървърни действия – функции, които се изпълняват на сървъра. Те могат да бъдат извикани както на клиента, така и по време на сървърното зареждане на страниците. Те са нова част от NextJS версия 13.3

- *app* – съдържат се всички страници, шаблони (layout.js / template.js), страници за грешка и всички останали специални файлове за структурата на NextJS уеб приложението.
- *components* – съдържат се най-малките и преизползваеми React компоненти в приложението.
- *configs* – съдържат се конфигурационни файлове.
- *constants* – съдържат се константни променливи.
- *context* – съдържат се глобални обвивки на проекта – React Context.
- *errors* – съдържат се специални / разширени (custom) грешки
- *fetch* – съдържат се всички спомагателни функции за извличане на данни.
- *hooks* – съдържат се всички преизползваеми React State функции.
- *messages* – съдържат се речниците, използвани от next-intl, за интернационализация.
- *partials* – съдържат се т.нар. „части“, които са логически разделени компоненти, които са по-големи от обикновените компоненти и обикновено съдържат други компоненти в себе си.
- *schemas* – съдържат се схемите, използвани от Zod библиотеката, за проверка на съдържанието.
- *styles* – съдържат се всички CSS / SCSS стилове за даден NextJS проект, тъй като този в този проект се използва само TailwindCSS, в тази директория се съдържа само „global.css” файл, който се използва с конфигурационни цели.
- *types* – съдържат се всички създадени типове, които се използват в приложението.

- *utils* – съдържат се всички спомагателни функции, които се използват често в уеб приложението.
- *wrappers* – съдържат се компоненти-обвивки, които се използват за обвивка на други компоненти



фиг. 3.2.2. – Файлова структура на клиентския софтуер

3.2.2. Конфигуриране на връзка с приложно-програмния интерфейс

С цел консистентно използване на URL адрес за приложно-програмния интерфейс, е създаден конфигурационен файл (/configs/api.ts) , който извлича от “.env” конфигурационния файл URL адреса на приложно-програмния интерфейс.

```
export const API_URL =
  process.env.NEXT_PUBLIC_API_URL || 'http://localhost:8080/api';
```

фиг. 3.2.3. – Конфигурационен файл за URL на ППИ

Освен това в някои клиентски компоненти се използва Axios библиотеката с или без SWR, вместо вградената и модифицирана от NextJS fetch() функция. За използване на една инстанция на Axios е създаден и Axios конфигурационен файл (/configs/axios.ts), в който се дефинира и адреса на приложно-програмния интерфейс.

```
import axios, { AxiosInstance } from 'axios';
import { API_URL } from '@/configs/api';

export const axiosInstance: AxiosInstance = axios.create({
  baseURL: API_URL,
});
```

фиг. 3.2.4. – Конфигурационен файл за Axios библиотеката

Тъй като SWR приема функция за извличане на данни, е създадена специална функция за извличане на данни, която използва Axios инстанцията.

```
import { AxiosResponse } from 'axios';
import { axiosInstance } from '@/configs/axios';

const fetcher = async <DataType>(url: string): Promise<DataType> => {
  const response: AxiosResponse<DataType> = await axiosInstance.get(url);
  return response.data;
};

export default fetcher;

```

фиг. 3.2.5. – Специална (custom) функция за извличане на данни за SWR

3.2.3. Обработка на клиентски и сървърни грешки

Една от най-важните части на едно клиентско приложение, освен сигурността, е предоставянето на грешки на потребителя по подходящ начин. Затова в този проект са създадени няколко помощни функции, от

които някои са комбинирани с библиотеката Zod за клиентско потвърждение и верифициране на информация, докато други са обвързани с обработката на грешки от сървъра.

Първо ще бъде разгледана обработката на сървърни грешки, която се използва за обработка на специални грешки, върнати от сървъра или за автоматични валидационни грешки от "class-validator" на NestJS.

Създадени са помощни типове за формата на грешките, получени от сървъра – „ValidationError" и „ErrorResponse", както и резултата за форматираните и извлечени сървърни грешки – „ExtractedServerErrors".

```
type ValidationError = {
  constraints: {
    [key: string]: string;
  };
  property: string;
};

type ValidationErrorsResponse = {
  errors: ValidationError[];
};

type ErrorResponse = {
  code: string;
  message: string; // field in this case
};

export type ExtractedServerErrors = {
  errors: {
    [key: string]: string[];
  };
} | null;
```

фиг. 3.2.6. – Помощни типове за извличането на сървърни грешки

Сървърните грешки, които се водят като познати, са разделени на два вида според техния код на грешка: 400 – грешно подадена заявка, или 409 – конфликт на записи. При статус грешно подадена заявка, означава, че данните, които са подадени, са грешни. Следователно се очаква автоматична грешка от „class-validator" на NestJS. Ако пък е конфликтна грешка – това е специална грешка, която е вдигната или от кода на проекта, или от Prisma и обработена през NestJS специалния

интерсептор. Идеята на обработените грешки е да бъдат с единен формат от главен обект с обекти, разделени на грешки – масив от стрингове.

```
const extractServerErrors = async (
  res: Response,
  t?: (key: string) => string,
): Promise<ExtractedServerErrors> => {
  if (res.status === 400) {
    const body = (await res.json()) as ValidationErrorsResponse;

    const serverErrors = body.errors.reduce(
      (acc, { constraints, property }) => ({
        ...acc,
        [property]: Object.keys(constraints).map((key) =>
          t ? t(`errors.server.${key}`) : key,
        ),
      }),
      {},
    );

    return {
      errors: {
        ...serverErrors,
      },
    };
  }

  if (res.status === 409) {
    const body = (await res.json()) as ErrorResponse;

    return {
      errors: {
        [body.message]: [t ? t(`errors.server.${body.code}`) : body.code],
      },
    };
  }

  return null;
};
```

фиг. 3.2.7. – Функция за извличане (с опционално превеждане) на сървърни грешки

Обаче преди да се стигне до сървърни грешки, първо се преминава през клиентски такива, за се спестят заявки до сървъра и за по-бърза обратна връзка с потребителя.

С тази цел са създадени функциите: *formatErrors()*, *extractErrors()* и *extractTranslatedErrors()*. Първата функция – *formatErrors()*, се използва преди извикване на която и да е функция за извличане на клиентски грешки. Използва се за форматиране на грешките от Zod библиотеката, за по-лесна обработка от извличащите функции. В нея се извиква

вградената *.format()* функция, която се използва при грешките, като след това им се поставя тип за ключ с обект с масив от стрингове.

```
import { SafeParseError } from 'zod';

export const formatErrors = (result: SafeParseError<unknown>) =>
  result.error.format() as unknown as {
    [key: string]: {
      _errors: string[];
    };
  };

```

фиг. 3.2.8. – Функция за форматиране на грешките от Zod

За функциите за извличане на грешки са необходими и помощни типове: „ErrorObject" – за форматираната грешка (вход) и „ExtractedErrors" – за изходните грешки, които са под единен формат.

```
interface ErrorObject {
  _errors?: string[];

  [key: string]: ErrorObject | string[] | undefined;
}

type ExtractedErrors = {
  [key: string]: string[] | ExtractedErrors;
};
```

фиг. 3.2.9. – Тип за форматирана клиентска грешка от Zod

И двете функции за извличане на грешки са рекурсивни, тъй като някои полета, може да имат повече от една проверка или деца.

```
export const extractErrors = (errors: ErrorObject): ExtractedErrors =>
  Object.keys(errors).reduce<ExtractedErrors>((acc, key) => {
    const errorEntry = errors[key];

    if (key !== '_errors' && errorEntry) {
      if (
        '_errors' in errorEntry &&
        Array.isArray(errorEntry._errors) &&
        errorEntry._errors.length > 0
      ) {
        if (errorEntry._errors.length > 0) {
          acc[key] = errorEntry._errors;
        }
      } else {
        const subErrors = extractErrors(errorEntry as ErrorObject);
        if (Object.keys(subErrors).length > 0) {
          acc[key] = subErrors;
        }
      }
    } else if (key === '_errors' && Array.isArray(errorEntry)) {
      acc['root'] = errorEntry;
    }
    return acc;
  }, {});
```

фиг. 3.2.10. – Функция за извличане на Zod клиентски грешки в общ обект

```
export const extractTranslatedErrors = (
  formattedErrors: ErrorObject,
  t: (key: string) => string,
): ExtractedErrors => {
  const errors = extractErrors(formattedErrors);
  return translateErrors(errors, t);
};

export const translateErrors = (
  errors: ExtractedErrors,
  t: (key: string) => string,
) => {
  const translatedErrors: ExtractedErrors = {};

  Object.keys(errors).forEach((key) => {
    const currentError = errors[key];
    if (Array.isArray(currentError)) {
      translatedErrors[key] = currentError.map((error: string) => t(error));
    } else {
      translatedErrors[key] = translateErrors(currentError, t);
    }
  });

  return translatedErrors;
};
```

фиг. 3.2.11. – Функция за превеждане на извлечените Zod клиентски грешки

3.2.4. Създаване на потребителски профили, удостоверение и авторизация. Защита на страници и действия.

Тъй като от NextJS 13 и нагоре има два вида компоненти – клиентски и сървърни, те по някакъв начин трябва да имат споделено пространство за съхранение на данни. В NextJS 13.3 излиза нова функционалност, наречена „Server Actions" – сървърни действия. Това са функции, които се изпълняват на сървъра. Те позволяват и за споделено пространство за съхранение на бисквитки, които се перфектни за съхранение на "access" и "refresh" токените, които се използват за авторизация след успешна регистрация или вход.

За да дадена функция да бъде сървърна, тя трябва или да започва с: „use server", или самият (ако са в отделен файл от компонент) да започва (подобно на клиентските компоненти – „use client") с „use server".

За управлението на бисквитките за токените са създадени следните помощни сървърни функции: *setTokens()* – за записване или

презаписване на бисквитки за двата токена, и *getTokens()* – за извличане на двата токена от бисквитките.

Една от най-важните функции, извикани в тази помощна сървърна функция, е *noStore()*, тъй като тя премахва кеширането, което е изключително агресивно с NextJS 13 и 14. След това се декодират токените, за да се прочете докога са валидни. Което ще позволи, при създаването на бисквитките, да се постави техен срок на годност.

```
const setTokens = (accessToken: string, refreshToken?: string | null) => {
  noStore();

  const cookieStore = cookies();

  const accessExpiry = jwtDecode(accessToken).exp;
  cookieStore.set('access_token', accessToken, {
    expires: accessExpiry ? accessExpiry * 1000 : undefined,
  });

  if (!refreshToken) return;

  const refreshExpiry = jwtDecode(refreshToken).exp;
  cookieStore.set('refresh_token', refreshToken, {
    expires: refreshExpiry ? refreshExpiry * 1000 : undefined,
  });
};
```

фиг. 3.2.12. – Помощна сървърна функция за записване на токените за сигурност в бисквитките

Другата функция е *getTokens()*, при нея също се достъпва хранилището за бисквитки и се изземат двете бисквитки, като се връща обект с токените.

```
const getTokens = () => {
  const cookieStore = cookies();
  const accessToken = cookieStore.get('access_token');
  const refreshToken = cookieStore.get('refresh_token');

  return {
    accessToken: accessToken?.value ?? undefined,
    refreshToken: refreshToken?.value ?? undefined,
  };
};
```

фиг. 3.2.13. – Помощна сървърна функция за извличане на токените за сигурност от бисквитките

За влизането на потребител в профил са създадени две сървърни функции – една главна и една помощна.

В помощната сървърна функция за влизане на потребител се прави заявка и се обработват нейните данни. При успех на заявката се извличат токените за сигурност от тялото на отговора и се връщат, както и запазват в хранилището за бисквитки. Докато при неуспех се връща грешка за невалидни данни за вход.

```
const login = async (email: string, password: string): Promise<boolean> => {
  noStore();

  const res = await fetch(`${API_URL}/auth/login`, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      email,
      password,
    }),
  });

  if (res.status === 201) {
    const { access_token, refresh_token } = (await res.json()) as AuthResponse;

    setTokens(access_token, refresh_token);
    return true;
  }

  throw new FormError('errors.invalid_credentials');
};
```

фиг. 3.2.14. – Помощна функция за влизане на потребител в профил

Главната сървърна функция (сървърно действие) за влизане на потребител също започва с извикване на функцията noStore() и се задава променлива за състояние на успешно или неуспешно влизане на потребителя, след което се прави проверка чрез Zod за грешки във входните данни от формата за влизане. При успешно преминаване на клиентската валидация се извиква помощната функция в блок за събиране на грешки (try-catch блок). Тъй като по стандарт не може да се извиква redirect() функцията на NextJS в try-catch блок, е създадена гореспоменатата булева променлива. При успешен опит за влизане не само се препраща потребителя към главната страница, но той също бива ревалидиран от NextJS кеша. При връщане на позната грешка – се връща

нейното наименование, докато при непозната грешка се връща обратна връзка – сървърна грешка 500.

```
const loginAction = async (prevState: unknown, formData: FormData) => {
  noStore();

  let logged = false;

  const validatedData = loginSchema.safeParse({
    email: formData.get('email'),
    password: formData.get('password'),
  });

  if (!validatedData.success) {
    return {
      messages: validatedData.error.issues.map((issue) => issue.message),
    };
  }

  const { email, password } = validatedData.data;

  try {
    logged = await login(email, password);
    revalidateTag('user');
  } catch (err) {
    if (err instanceof FormError) {
      return {
        messages: [err.message],
      };
    }

    return {
      messages: ['errors.500'],
    };
  }

  if (logged) redirect('/');
};
```

фиг. 3.2.15. – Главна сървърна функция за влизане на потребител в профил

Това действие се конфигурира в клиентски компонент с *useFormState()*, за да се следи прогрес и грешки, върнати от функцията. Тя бива поставена като action свойство на JSX формата.

```
const [state, formAction] = useFormState(loginAction, initialState);
const t = useTranslations('auth.login');

return (
  <form
    action={formAction}
    className='flex w-full flex-col gap-4 font-semibold sm:w-fit'
  >
      ...
  </form>
);
```

фиг. 3.2.16. – Употреба на сървърните функции за форма

Сървърната функция за регистрация е по-особена, тъй като при нея не се извършва клиентска валидация или връщане на обработени сървърни грешки, поради факта, че тези грешки вече са били проверени в предишните етапи на регистрация. Подобно на помощната сървърна функция за вход на потребител, тук само се прави заявка и се връща или за успех – токени, или се връща грешка.

```
const signup = async (data: SignupFormState) => {
  try {
    const res = await fetch(`${API_URL}/auth/signup`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify(data),
    });

    if (res.ok) {
      const { access_token, refresh_token } =
        (await res.json()) as AuthResponse;

      setTokens(access_token, refresh_token);
      return true;
    }

    return {
      messages: ['errors.500'],
    };
  } catch (e) {
    return {
      messages: ['errors.500'],
    };
  }
};
```

фиг. 3.2.17. – Сървърна функция за регистрация на потребител

Защитите на страниците са имплементирани както на ниво страница, така и още преди изписването на самата страница, на ниво заявка, чрез междинния софтуер на NextJS – middleware.ts.

```
// === Pages where user should not be logged in ===

if (
  NOT_LOGGED_IN_PAGES.some((page) => path.startsWith(page)) &&
  (request.cookies.get('access_token') ||
    response.cookies.get('access_token'))
) {
  return NextResponse.redirect(new URL('/', request.url));
}

// === Pages where user should be logged in ===

if (
  LOGGED_IN_PAGES.some((page) => path.startsWith(page)) &&
  !(
    request.cookies.get('access_token') ||
    response.cookies.get('access_token')
  )
) {
  return NextResponse.redirect(new URL('/auth/login', request.url));
}
```

фиг. 3.2.18. – Проверка за регистрация на потребител и достъпността му

Друга функция, която е сървърна, е за проверката дали потребител има права за достъп до определени модифициращи функции. Тази функция се използва най-често за показване на бутони за модифициране / изтриване, както и за достъп до техните страници. В нея се сравняват ИН на създателя на дадена публикация и ИН на влезналия потребител, ако има такъв, както се прави и проверка за ролята на потребителя.

```
const canModify = async (userUuid: string) => {
  noStore();

  const accessToken = await getAuth('client');

  if (!accessToken) {
    return false;
  }

  const res = await fetch(`${API_URL}/user/me`, {
    headers: {
      Authorization: accessToken,
    },
  });

  if (!res.ok) {
    return false;
  }

  const data = (await res.json()) as {
    uuid: string;
    role: 'ADMIN' | 'MOD' | 'USER';
  };

  return data.uuid === userUuid || data.role === 'ADMIN' || data.role === 'MOD';
};
```

фиг. 3.2.19. – Сървърна функция за проверка за права за достъп за модифициране на даден обект

Друг пример за права за достъп е този за правата за гледане, който не е функция, а част от заявката за извличане на обява или място. Те могат да бъдат разгледани подробно в точки 3.2.9. и 3.2.11..

Една от най-важните части в сигурността на приложението и правилната употреба на токените са обработката на токени след изтичане на „access” токена и неговото подновяване с „refresh” токена. Това се изпълнява от хибридната (сървърна, но се използва и като нормална функция в междинния софтуер) функция – *refreshTokensAndReturnAccess()*. В нея се достъпва хранилището за

бисквитки, като се извлича „refresh” токен, с който се извиква заявка до приложно-програмния интерфейс за подновяване на „access” токена. При успешно изпълнение на заявката се връща нов „access” токен, докато при неуспешна, може да се върнат две грешки – при 401 – специална грешка за изтекъл „refresh” токен, докато другата е обща.

```
export const refreshTokensAndReturnAccess = async () => {
  noStore();

  const cookieStore = cookies();
  const refreshToken = cookieStore.get('refresh_token');

  if (!refreshToken) {
    throw new Error('No refresh token found');
  }

  const response = await fetch(`${API_URL}/auth/refresh`, {
    method: 'GET',
    headers: {
      Authorization: `Bearer ${refreshToken.value}`,
      'Content-Type': 'application/json',
    },
  });

  if (response.status !== 200) {
    if (response.status === 401) {
      throw new HTTPUnauthorizedException();
    }

    throw new Error('Failed to refresh tokens');
  }

  const data = (await response.json()) as {
    access_token: string;
  };

  return data.access_token;
};
```

фиг. 3.2.20. – Функция за подновяване и връщане на нов „access” токен

Тази функция се извиква на две места – или в междинния софтуер, или по време на заявка – от помощните функции *isAuth()* и *getAuth()*.

isAuth() се използва за взимане на „access” токена и връщане дали потребителят е влязъл в профил. Важното на тази функция е, че приема аргумент дали се изпълнява по време на извличане на данни от клиент, или по време на сървърно извличане на данни, тъй като според документацията на NextJS не може да се използва хранилището за бисквитки в режим записване по време на сървърно извличане на данни.

Обаче при тях това не е нужно, заради подновяването на токените преди тяхното изпълнение – в междинния софтуер. Във функцията се взима „access” токена и проверява дали има такъв. Ако има – връща стойността му, ако няма – се опитва да го поднови.

```
export const isAuth = async (
  rendering: 'client' | 'ssr',
): Promise<string | false> => {
  noStore();

  const cookieStore = cookies();
  const accessToken = cookieStore.get('access_token');

  if (!accessToken) {
    try {
      const new_access = await refreshTokensAndReturnAccess();

      if (rendering === 'client') {
        setTokens(new_access);
      }

      return new_access;
    } catch (err) {
      return false;
    }
  }

  return accessToken.value;
};
```

фиг. 3.2.21. – Помощна функция за проверка дали потребител е регистриран

Другата помощна функция надгражда isAuth(), като връща задължително токен с „Bearer “ пред него, с цел да се използва веднага за заявки, или празна стойност.

```
export const getAuth = async (
  rendering: 'client' | 'ssr',
): Promise<string | null> => {
  noStore();

  const auth = await isAuth(rendering);

  if (!auth) {
    return null;
  }

  return `Bearer ${auth}`;
};
```

фиг. 3.2.22. – Помощна функция за запълване на Authorization хедър за заявка

Последното място, където се използва функцията за подновяване на „access” токена е в междинния софтуер, където се прави проверка за съществуващ „refresh” токен и несъществуващ „access” токен, където се

извиква помощна функция за подновяване токените с хранилището за бисквитки на самата заявка от междинния софтуер. При каквато и да е грешка - 4XX, се изтриват всички бисквитки за сигурност.

```
if (
  request.cookies.get('refresh_token') &&
  !request.cookies.get('access_token')
) {
  try {
    await refreshTokensMiddleware(response.cookies);
  } catch (err) {
    if (err instanceof HTTPUnauthorizedException) {
      deleteTokensMiddleware(response.cookies);
    } else {
      throw err;
    }
  }
}
```

фиг. 3.2.23. – Проверка за нужда от подновяване на „access" токена

Във функцията се подновява токена и се създава нова бисквитка, а във функцията за изтриване – те се изтриват от подаденото хранилище за бисквитки от междинния софтуер.

```
export const refreshTokensMiddleware = async (cookieStore: ResponseCookies) => {
  const newToken = await refreshTokensAndReturnAccess();
  const newTokenExpiry = jwtDecode(newToken).exp;

  cookieStore.set('access_token', newToken, {
    expires: newTokenExpiry ? newTokenExpiry * 1000 : undefined,
  });
};

const deleteTokensMiddleware = (cookieStore: ResponseCookies) => {
  cookieStore.delete('access_token');
  cookieStore.delete('refresh_token');
};
```

фиг. 3.2.24. – Помощни функции за сигурност за междинния софтуер

Последната функция за сигурност в това уеб приложение е тази за излизане на потребител от акаунт. Тази функция също се изпълнява в междинния софтуер, при отиване на адреса за излизане от профил. В нея се изземат токени, извиква се заявка на приложно-програмния интерфейс за излизане на потребителя и при валиден „refresh" токен, тя

бива успешно изпълнена и токенът е вече в черния списък, като след това се изтриват всички бисквитки за сигурност от уеб приложението. Те също се изтриват и при изтекъл „refresh” токен.

```
async function logout(response: NextResponse) {
  noStore();

  const { accessToken, refreshToken } = getTokens();

  if (!refreshToken || !accessToken) {
    deleteTokensMiddleware(response.cookies);
    return;
  }

  const res = await fetch(`${API_URL}/auth/logout`, {
    method: 'POST',
    headers: {
      Authorization: `Bearer ${refreshToken}`,
      'Content-Type': 'application/json',
    },
    body: JSON.stringify({
      accessToken,
      refreshToken,
    }),
  });

  if (res.status === 200 || res.status === 201 || res.status === 401) {
    deleteTokensMiddleware(response.cookies);
  }
}
```

фиг. 3.2.25. – Функция за излизане на потребителя от профил

3.2.5. Настройки на приложението – интернационализация (i18n) и теми

Глобалните обвивки на приложението се поставят в глобалния шаблон – layout.js файл. Тези контексти (обвивки) се слагат над децата на шаблона. В глобалния layout.js се приема параметър от URL адреса за език, който да бъде използван в приложението. Ако такъв параметър не е подаден, страницата ще върне 404 грешка. След това се извиква функцията за зареждане на всички съобщения (всички стрингове за всички езици), които след това се подават на *NextIntlClientProvider* за създаване на контекста. Под него се инициализира и контекстът за тема на приложението – *ThemeProvider*. Освен това, за интернационализация

се слага атрибут за език на *html* тага, както и се генерират всички статични пътища за всеки един език с *generateStaticParams()* функцията.

```
export function generateStaticParams() {
  return locales.map((locale) => ({ locale }));
}

export default function LocaleLayout({
  children,
  params: { locale },
}: Readonly<{
  children: React.ReactNode;
  params: { locale: string };
}>) {
  if (!locales.includes(locale)) {
    notFound();
  }

  unstable_setRequestLocale(locale);

  const messages = useMessages();

  return (
    <html lang={locale} className='dark'>
      <body
        className={`${inter.className} flex flex-row bg-background text-text`}
      >
        <NextIntlClientProvider messages={messages}>
          <ThemeProvider>
            <ModalProvider>
              <Navigation />
              <main className='flex min-h-screen w-full justify-center px-4 pb-4 pt-24 md:py-4'>
                {children}
              </main>
            </ModalProvider>
          </ThemeProvider>
        </NextIntlClientProvider>
      </body>
    </html>
  );
}
```

фиг. 3.2.26. – Част от глобалния шаблон на уеб приложението – layout.js

В изходния код на приложението има два конфигурационни файла – в единия се задават и връщат всички налични езици, както и този по подразбиране. В главния конфигурационен файл се обхождат и вкарват в приложението всички файлове, които съдържат съобщения за всеки език.

```
export const i18n = {
  defaultLocale: 'en',
  locales: ['en', 'bg'],
};

export type Locale = (typeof i18n)['locales'][number];
```

фиг. 3.2.27. – Конфигурационен файл за i18n

```
import { getRequestConfig } from 'next-intl/server';

export default getRequestConfig(async ({ locale }) => ({
  messages: (await import(`@/messages/${locale}.json`)).default,
}));

```

фиг. 3.2.28. – Главен конфигурационен файл за i18n

Тъй като вече приложението е конфигурирано да изисква задължително URL адресите да съдържат параметър за език, в междинния софтуер се конфигурира да служи за интернационализация с *createMiddleware()*, където са подадени всички езици, както и този по-подразбиране. Този междинен софтуер се използва по подразбиране, освен ако не се стигне до изпълнението на функциите в т. 3.2.4.. Във файла за междинния софтуер също се връща и конфигурация, която позволява на пътищата за статични файлове на NextJS да не им се поставя параметър за език.

```
const intl = createMiddleware({
  locales,
  defaultLocale: 'en',
});

export default async function middleware(request: NextRequest) {
  const response = intl(request);
  ...
}

export const config = {
  // Skip all paths that should not be internationalized
  matcher: ['/((?!_next|.*\\..*).*)'],
};
```

фиг. 3.2.29. – Междинен софтуер за интернационализация (i18n)

Една важна функционалност за потребителското изживяване е възможността на потребителя да разполага с избор на светла и тъмна тема на приложение, спрямо ръчна настройка (светла или тъмна тема),

или автоматичен избор, който се синхронизира с операционната система. В контекста се съдържат тема и функция за промяна на темата.

```
'use client';

import React, { createContext, useEffect, useState } from 'react';

export type Theme = 'light' | 'dark' | 'system';

const ThemeContext = createContext<{
  theme: Theme | undefined;
  changeTheme: (theme: Theme) => void;
}>({
  theme: 'system',
  changeTheme: () => {},
});
...
```

фиг. 3.2.30. – Създаване на контекст за определяне на тема на приложението

За да може контекстът да бъде достъпван от клиентски функции, той трябва да има специално създадена за него компонента функция (React Hook), която в себе си да извиква контекста и да вземе стойностите, които връща – темата и функцията за промяна на темата.

```
...
export const useTheme = () => {
  const { theme, changeTheme } = React.useContext(ThemeContext);

  return {
    theme,
    changeTheme,
  };
};
```

фиг. 3.2.31. – Създаване на компонента преизползваема функция (hook) за четене и модифициране на темата

В обвивката са дефинирани както и стойността на темата – чрез React State, така и функцията за промяна на темата. Освен това има две функции за цикъла на живота на обвивката (useEffect()), които се извикват при начално зареждане на приложението и при промяна темата те се използват за поставяне на класовете на избраната тема. Тази обвивка е разгледана на следващата страница *във фиг. 3.2.32.*.

```
export const ThemeProvider = ({ children }: { children: React.ReactNode }) => {
  const [theme, setTheme] = useState<Theme | undefined>(undefined);

  const changeTheme = (theme: Theme) => {
    const root = window.document.documentElement;

    switch (theme) {
      ...
    }
  };

  const getInitialTheme = (): Theme => {
    if (typeof window !== 'undefined' && window.localStorage) {
      const storedPrefs = window.localStorage.getItem('theme');
      if (typeof storedPrefs === 'string') {
        return storedPrefs as Theme;
      }
    }

    return 'system';
  };

  useEffect(() => {
    setTheme(getInitialTheme());
  }, []);

  useEffect(() => {
    const root = window.document.documentElement;

    switch (theme) {
      ...
    }
  }, [theme]);

  return (
    <ThemeContext.Provider
      value={{
        theme,
        changeTheme,
      }}
    >
      {children}
    </ThemeContext.Provider>
  );
};

export default ThemeContext;
```

фиг. 3.2.32. – Част от обвивката (провайдър) за тема

3.2.6. Извличане на данни по страници (Pagination)

Функциите за извличане на данни по страници с лимити са основна част от приложението, тъй като обяви, места, коментари, ревюта и множество от потребители се извличат точно по такъв начин. За тази цел е създадена помощна обвиваща функция на *fetch()* за извличане на данни, както и преизползваем компонент за промяна и управление на страници и лимити.

Помощната функция приема URL адрес, номер на страница, лимит, както и често използвани опции за заявката – кеширащи опции, опции за NextJS fetch() и хедъри на заявката. Във функцията се създава URL адрес, в който първо се проверява дали има query параметри и спрямо това дали да се постави въпросителен знак на URL адреса. След което се добавят параметрите за страница и лимит, както и в заявката се добавят останалите опции и настройки. След това се проверява дали нейното изпълнение е било успешно, като също се и правят проверки за права за достъп (401 и 403 - забранен достъп).

```
export const paginatedFetch = async <T>(
  url: string,
  page: number,
  limit: number,
  options?: {
    headers?: HeadersInit;
    cache?: RequestCache;
    next?: NextFetchRequestConfig;
  },
): Promise<PaginatedResponse<T>> => {
  const response = await fetch(
    `${API_URL}${url}${url.includes('?') ? '&' : '?'}page=${page}&limit=${limit}`,
    {
      ...options,
    },
  );

  console.warn('url', `${API_URL}${url}?page=${page}&limit=${limit}`);

  if (!response.ok) {
    switch (response.status) {
      case 401:
        throw new HTTPUnauthorizedException();
      case 403:
        throw new HTTPForbiddenException();
      case 404:
        return {
          data: [],
          totalCount: 0,
        } as unknown as PaginatedResponse<T>;
    }

    throw new Error('Failed to fetch data', {
      cause: response.status,
    });
  }

  return (await response.json()) as PaginatedResponse<T>;
};
```

фиг. 3.2.33. – Помощна функция за извличане на данни по страници с лимити

Помощната функция paginatedFetch() е TypeScript Generic функция, която връща отговор от тип PaginatedResponse, който също е TypeScript

Generic, тъй като данните, които връща са лимит, номер на страница, общ брой елементи и самите данни, които са Generic – тип, който се подаде.

```
export type PaginatedResponse<T> = {
  data: T[];
  limit: number;
  page: number;
  totalCount: number;
};
```

фиг. 3.2.34. – Generic тип за данни за отговор на заявка със страници и лимити

В клиентските компоненти се използват вградените функции за извличане на данни от URL адреса на приложението – път и параметри. При промяна на страница се сменят параметрите на URL адреса чрез помощната функция – *createQueryString()*. Ако страницата надвишава последната страница, то параметъра се променя на последната.

```
export const PageOptions = ({ page, limit, totalItems }: PageOptionsProps) => {
  const router = useRouter();
  const pathname = usePathname();
  const searchParams = useSearchParams();
  const totalPages = Math.ceil(toInteger(totalItems) / toInteger(limit));

  const createQueryString = useCallback(
    (name: string, value: string) => {
      const params = new URLSearchParams(searchParams.toString());
      params.set(name, value);

      return params.toString();
    },
    [searchParams],
  );

  const handlePage = (nextPage: number) => {
    if (nextPage === page) return;

    router.replace(
      `${pathname}?${createQueryString('page', nextPage.toString())}`,
      {
        scroll: false,
      },
    );
  };

  if (page > totalPages) {
    router.replace(
      `${pathname}?${createQueryString('page', totalPages.toString())}`,
      {
        scroll: false,
      },
    );
  }

  return (
    ...
  );
};

export default PageOptions;
```

фиг. 3.2.35. – Клиентски компонент за управление на страници и лимити

3.2.7. Конфигурация на Google Maps

Подобно на всички останали конфигурационни файлове, този на Google Maps също извлича от конфигурационния ".env" файл ИН и ключ за достъп до приложно-програмния интерфейс на Google Maps.

```
const GMAPS_API = process.env.NEXT_PUBLIC_GOOGLE_MAPS_API_KEY ?? '';
const GMAPS_ID = process.env.NEXT_PUBLIC_GOOGLE_MAPS_ID ?? '';

export { GMAPS_API, GMAPS_ID };
```

фиг. 3.2.36. – Извличане на ключа и ИН за достъп на ППИ на Google Maps

Първоначално за обратното геокодиране на Google Maps бе използвана библиотеката „react-geocode", но тъй като тя не разполага с възможността да се подават референти URL адреси, тя беше премахната от проекта и заменена с ръчно написана компонентна функция (hook). Референтните URL адреси позволяват да се използват вградените функции за защита на Google Maps ППИ, затова е създадена и помощната функция за зареждане на техния ППИ, заедно с ключа.

```
const loadScript = async (): Promise<void> => {
  const loader = new Loader({
    apiKey: GMAPS_API,
    version: 'weekly',
  });

  await loader.load();
};
```

фиг. 3.2.37. – Зареждане на ППИ на Google Maps
за използване на обратно геокодиране

Помощната функция, която е част от *useAddress()*, *getAddressFromApi()* се стартира със зареждането на ППИ на Google Maps, след което създава инстанции на *Geocoder* класа за обратно геокодиране (от подадените координати) и *LatLang* (координати от 2 числа), след което

прави заявка към ППИ и при успешен резултат задава стойността на компонентната функция на адреса.

```
const useAddress = (lat: number, lng: number, language: string): string => {
  const [address, setAddress] = useState('Loading...');

  const getAddressFromApi = async (
    lat: number,
    lng: number,
    language = 'en',
  ): Promise<google.maps.GeocoderResult> => {
    await loadScript();

    const geocoder = new google.maps.Geocoder();
    const latlng = new google.maps.LatLng(lat, lng);

    return new Promise((resolve, reject) => {
      try {
        void geocoder.geocode(
          {
            location: latlng,
            language,
          },
          (results, status) => {
            if (status === google.maps.GeocoderStatus.OK) {
              if (
                results &&
                results.length > 0 &&
                results[0].formatted_address
              ) {
                resolve(results[0]);
              } else {
                reject(new Error('Couldnt find the address'));
              }
            } else {
              reject(new Error(`Geocoder failed due to: ${status}`));
            }
          },
        );
      } catch (err) {
        reject(err);
      }
    });
  };

  useEffect(() => {
    const getAddress = async () => {
      try {
        const res = await getAddressFromApi(lat, lng, language);
        setAddress(res.formatted_address);
      } catch (err) {
        console.error(err);
      }
    };

    void getAddress();
  }, [lat, lng, setAddress]);

  return address;
};
```

фиг. 3.2.38. – Компонента преизползваема функция (hook) за използване на обратно геокодиране чрез ППИ на Google Maps

3.2.8. Извличане на обяви и коментари

За извличането на обяви е създадена обща спомагателна функция, която се използва навсякъде, където се взимат няколко обяви наведнъж – на началната страница, при изброяването на всички обяви, при показването на категории, както и в профилите на потребителите (публични или собствени).

Функцията за извличане на много обяви приема номер на страница, лимит, потребителско име, ИН на категория и токен за достъп. Всички тези параметри са незадължителни. След това се създава URL адрес, спрямо какво е подадено от незадължителните параметри, както и се подават номера на страница и лимита на специалната функция за извличане на данни – *paginatedFetch()*. Освен това се подават няколко NextJS опции за ревалидиране през даден интервал и опционално – токен за авторизация, ако се прави опит за достъпване на собствени потребителски обяви.

```
export const getListings = async (
  page: number = 1,
  limit: number = 6,
  username?: string,
  category?: string,
  auth?: string,
) => {
  try {
    return await paginatedFetch<ShortListingResponse>(
      `/${username ? `user/${username}/` : ''}listing${
        category ? `?category=${category}` : ''
      }`,
      page,
      limit,
      {
        next: {
          revalidate: 60,
        },
        headers: {
          Authorization: auth || '',
        },
      },
    );
  } catch (error) {
    console.error(error);
    return null;
  }
};
```

фиг. 3.2.39. – Извличане на обяви

Следващата функция за извличане на данни е тази за извличане на една обява, тук се приема само един аргумент – ИН на обява. В нея се извиква обикновената *fetch()* функция, като се подава токен за авторизация при нужда, ако обявата е частна. При код за липса на достъп и права, се връща грешка, която след това се използва за показване на страница за забранен достъп.

```
const getListing = async (uuid: string) => {
  const auth = await getAuth('ssr');

  const response = await fetch(`${API_URL}/listing/${uuid}`, {
    cache: 'no-cache',
    headers: auth
      ? {
          Authorization: auth,
        }
      : {},
  });

  if (!response.ok) {
    switch (response.status) {
      case 401:
        throw new HTTPUnauthorizedException();
      case 403:
        throw new HTTPForbiddenException();
      case 404:
        return null;
    }

    throw new Error(`Couldn't fetch listing ${response.status}`);
  }

  return (await response.json()) as FullListingResponse;
};
```

фиг. 3.2.40. – Извличане на обява

Функцията за извличане на коментари е подобна на тази за обяви само че при нея се подават само ИН на обявата, страница и лимит.

```
const getComments = async (
  listingUUID: string,
  page: number = 1,
  limit: number = 12,
) => {
  const auth = await getAuth('ssr');

  return paginatedFetch<Comment>(
    `/listing/${listingUUID}/comment`,
    page,
    limit,
    {
      next: {
        tags: [`/listing/${listingUUID}/comment`],
      },
      headers: auth
        ? {
            Authorization: auth,
          }
        : {},
    },
  );
};
```

фиг. 3.2.41. – Извличане на коментари

3.2.9. Извличане на места и работа с интерактивна карта на Google Maps

Функцията за извличането на няколко места също се използва навсякъде – в началната страница, в категориите, при списъка от всички места, при профилните страници, както и при картата. Затова, по подобие на функцията за извличане на обяви, се подават страница, лимит, потребителско име, ИН на категория и токен за достъп. Като след това се създава заявка според подадените аргументи със специалната функция *paginatedFetch()*.

```
export const getPlaces = async (
  page: number = 1,
  limit: number = 4,
  username?: string,
  category?: string,
  auth?: string,
) => {
  try {
    return await paginatedFetch<CardPlace>(
      `${username ?
        `/user/
            ${username}` : ''}
            /place?format=card
            ${category ? `&category=${category}`
        : ''}`,
      page,
      limit,
      {
        next: {
          revalidate: 60,
        },
        cache: 'no-cache',
        headers: {
          Authorization: auth || '',
        },
      },
    );
  } catch (error) {
    console.error(error);
    return null;
  }
};
```

фиг. 3.2.42. – Извличане на места

За поставяне на компонент за визуализиране на Google Maps карта, той трябва да бъде обвит с контекста за достъп до приложно-програмния

интерфейс на Google Maps, като му се подаде като свойство – ключа за сигурност на приложно-програмния интерфейс.

```
      <APIProvider apiKey={GMAPS_API}>
        <div
          className='flex h-[calc(100vh-7rem)] w-full gap-4 md:h-[calc(100vh-2rem)]'
          ref={parent}
        >
          <MapsPartialPage uuid={uuid} />
          <Suspense fallback={<div>Loading...</div>}>{children}</Suspense>
        </div>
      </APIProvider>
    </SWRConfig>
```

фиг. 3.2.43. – Конфигуриране на обвивка за Google Maps карта

След това във всяко дете на *ApiProvider* контекста може да се постави Google Maps карта, като може да се добавят допълнителни конфигурационни настройки като ниво на приближеност, център на картата, контроли и ИН на картата.

```
<div className='h-full w-full overflow-hidden rounded-xl border border-stroke bg-primary'>
  <Map
    zoom={13}
    center={{
      lat: 42.6977,
      lng: 23.3219,
    }}
    fullscreenControl={false}
    gestureHandling={'greedy'}
    streetViewControl={false}
    mapTypeControl={false}
    mapId={GMAPS_ID}
  >
    <MarkersMap
      data={places}
      openPlace={uuid}
      setOpenPlace={setOpenPlace}
    />
  </Map>
</div>
```

фиг. 3.2.44. – Визуализиране на Google Maps карта

За да иконките на местата да не се наслагват една над друга и да не натоварват устройството, на което е заредено уеб приложението, се използва т.нар. клъстър на места, който комбинира места, които са на много близко разстояние, спрямо нивото на приближеност / отдалеченост на картата. Клъстерът се използва от библиотека на Google Maps, като при промяна на картата или данните (т.е. местата) се създава

нов клъстер за дадената карта. При промяна на нивото на приближеност се извикват функции за изчистване и добавяне на маркерите наново, като е важно да се отбележи, че всеки маркер има референция към него, който се поставя с функцията *setMarkerRef()*.

```
const cluster = useRef<MarkerClusterer | null>(null);

const setMarkerRef = (marker: Marker | null, key: string) => {
  if (marker && markers[key]) return;
  if (!marker && !markers[key]) return;

  setMarkers((prev) => {
    if (marker) {
      return {
        ...prev,
        [key]: marker,
      };
    } else {
      const newMarkers = { ...prev };
      delete newMarkers[key];
      return newMarkers;
    }
  });
};

useEffect(() => {
  if (!map || !data) return;
  if (!cluster.current) {
    cluster.current = new MarkerClusterer({ map });
  }
}, [map, data]);

useEffect(() => {
  if (!cluster.current) return;

  cluster.current.clearMarkers();
  cluster.current.addMarkers(Object.values(markers));
}, [markers]);
```

фиг. 3.2.45. – Разпределение и събиране на елементи на картата в клъстер

За всяко място се създава маркер с референция и функция при натискане да се отвори или затвори дадено място встрани на картата.

```
<AdvancedMarker
  key={place.uuid}
  position={{
    lat: place.lat,
    lng: place.lng,
  }}
  onClick={() => {
    if (openPlace === place.uuid) {
      setOpenPlace(null);
    } else {
      setOpenPlace(place.uuid);
    }
  }}
  ref={(marker) => {
    setMarkerRef(marker, place.uuid);
  }}
>
  <span className='text-5xl'>{place.icon}</span>
</AdvancedMarker>
```

фиг. 3.2.46. – Компонент за визуализация на място като маркер на картата

3.2.10. Извличане на място и ревюта

Извличането на място е създадено по подобие на извличането на единична обява. Подобно се извършват и проверки дали обявата е частна и дали потребителят има достъп до нея.

```
export const getPlace = async (uuid: string) => {
  const auth = await getAuth('ssr');

  const response = await fetch(`${API_URL}/place/${uuid}`, {
    cache: 'no-cache',
    headers: auth
      ? {
          Authorization: auth,
        }
      : {},
  });

  if (!response.ok) {
    switch (response.status) {
      case 401:
        throw new HTTPUnauthorizedException();
      case 403:
        throw new HTTPForbiddenException();
      case 404:
        return null;
    }

    throw new Error(`Couldn't fetch listing ${response.status}`);
  }

  return (await response.json()) as Place | null;
};
```

фиг. 3.2.47. – Извличане на място

Функцията за извличане на ревюта по страници също е изградена по подобие на тази за коментари на обяви.

```
const getReviews = async (
  uuid: string,
  page: number = 1,
  limit: number = 12,
) => {
  const response = await paginatedFetch<ReviewType>
(    `/place/${uuid}/review`,
    page,
    limit,
    {
      cache: 'no-cache',
    },
  );

  if (!response) {
    return null;
  }

  return response;
};
```

фиг. 3.2.48. – Извличане на ревюта

3.2.11. Качване на файлове в AWS S3

Първата стъпка за качване на файлове в S3 е извличането на удостоверен URL адрес за качване на файл в кофата. Тъй като качването на файлове е защитено, се изисква преминаването през приложно-програмния интерфейс за получаване на удостоверен URL адрес за качване, както и уникален ключ за самия файл. След успешно изпълнение на заявката, се очаква връщането на URL адрес за качване чрез POST заявка, ключ на файла, както и публичен URL адрес за достъпване на качения файл, който често се използва във функциите за качване на даден ресурс, като се подава URL вместо самата снимка.

```
export const getSignedUrl = async (
  file: File,
  acl: 'public-read' | 'private',
  token: string,
) => {
  const res = await fetch(
    `${API_URL}/file/upload?name=${file.name}&type=${
      file.type.split('/')[1]
    }&acl=${acl}`,
    {
      headers: {
        Authorization: token,
      },
    },
  );

  if (!res.ok) {
    throw new Error('Something went wrong');
  }

  return (await res.json()) as {
    url: string;
    key: string;
    public_url: string;
  };
};
```

фиг. 3.2.49. – Извличане на линк на удостоверен линк за качване в AWS S3

След получаване на удостоверен URL адрес за качване на снимка в AWS S3, може да се изпълни функцията за самото качване на файл в кофата – *uploadS3()*. За нея се подават файл и URL адрес, като след това се

изпълнява PUT заявка към удостоверения адрес на AWS S3 кофата. Помощната функция връща булева стойност – дали е изпълнена успешно.

```
export const uploadS3 = async (file: File, url: string) =>
{ const res = await fetch(url, {
    method: 'PUT',
    body: file,
    headers: {
      'Content-Type': file.type,
    },
  });

  return res.ok;
};
```

фиг. 3.2.50. – Функция за качване на файл в AWS 3 с PUT заявка

3.2.12. Създаване на място

Качването както на място, така и на обява се извършва изцяло на клиентско ниво, тъй като се изпълняват много валидации и същевременно качване на файлове. Те са разделени на 2 функции – помощна, която се изпълнява накрая – самата POST заявка за качване на дадения ресурс, и главната функция, която извършва цялата валидация и качване на снимки в AWS S3.

В помощната функция *createPlace()* се подават готови данни, токен за достъп и URL адреси на снимките. В нея се извършват форматирането на тялото и заявката за качване на мястото.

```
export async function createPlace(
  data: CreatePlaceForm,
  signedUrls: SignedUrlResponse[],
  token: string,
) {
  const body = {
    icon: data.icon,
    name: data.name,
    description: data.description,
    status: data.status,
    categoryUuid: data.category.uuid,
    tags: data.tags,
    services: data.services.map((service) => service.uuid),
    lat: data.location.lat,
    lng: data.location.lng,
    images: signedUrls.map((url) => url.public_url),
  };

  return fetch(`${API_URL}/place`, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      Authorization: token,
    },
    body: JSON.stringify(body),
  });
}
```

фиг. 3.2.51. – Помощна функция за извършване на заявка за качване на място

Главната функция приема всички данни от формата, снимките, както и получава клиентска функция за показване на грешки и преводи. Първата проверка, която се изпълнява, е за това дали потребителят е регистриран, след това се проверяват, заедно с помощните функции, обяснени в точка 3.2.3., дали данните от формата и снимките са валидни. След това се продължава с още няколко проверки, които не са показани на *фиг. 3.2.52.*, дали данните са празни.

```
export const handleCreatePlace = async (
  data: CreatePlaceForm,
  images: ImageInputProps[],
  handleError: (newErrors: Partial<CreatePlaceErrors>) => void,
  t: (key: string) => string,
) => {
  const token = await getAuth('client');
  if (!token) {
    handleError({ global: ['You must be logged in to create a listing'] });
    return;
  }

  const result = createPlaceSchema.safeParse(data);
  const imagesResult = createImagesSchema.safeParse(images);
  let newErrors: Partial<CreateProductErrors> = {};

  if (!result.success) {
    const formatted = formatErrors(result);
    newErrors = {
      ...newErrors,
      ...extractTranslatedErrors(formatted, t),
    };
  }

  if (!imagesResult.success) {
    const formatted = formatErrors(imagesResult);
    const extracted = extractTranslatedErrors(formatted, t);

    if (
      extracted['root'] &&
      Array.isArray(extracted['root']) &&
      extracted['root'].length
    ) {
      newErrors = {
        ...newErrors,
        images: extracted['root'],
      };
    }
  }
  ...
```

фиг. 3.2.52. – Първа част от функцията за качване на място – валидиране

След успешна клиентска проверка, функцията продължава, като първо взима удостоверени URL адреси за всяка снимка с помощна функция, която обхожда всяка снимка и създава Promise за нея. След успешно извличане на удостоверените за качване URL адреси, започва подобен процес, но за качване на файловете в AWS S3 кофата. След това се извършва проверка дали качването на снимките е било успешно, ако

не е било, се връща грешка на потребителя. Ако всичко дотук е било успешно, се извиква помощната функция, описана на *фиг. 3.2.52.* и се връща нейния отговор.

```
  const signedUrls = await getSignedUrls(images, data.status,
token);
  const uploadResults = await uploadImages(signedUrls, images);
  const finalErrors = uploadResults
    .filter((res) => !res)
    .map(() => 'Error uploading image');

  if (finalErrors.length) {
    handleError({ images: finalErrors });
    return;
  }

  const response = await createPlace(data, signedUrls, token);
  if (!response.ok) {
    const listing = (await response.json()) as {
      message: string;
    };
    handleError({ global: [listing.message] });
    return;
  }

  return (await response.json()) as Place;
};
```

фиг. 3.2.53. – Втора част от функцията за качване на място

3.2.13. Създаване на обява

Процесът по качване на обява е разделен по подобен начин на две функции. Помощната функция *createListing()* е разгледана на *фиг. 3.2.54.*.

```
export async function createListing(
  data: CreateListingForm,
  signedUrls: SignedUrlResponse[],
  token: string,
) {
  const body = {
    title: data.name,
    description: data.description,
    type: 'PRODUCT',
    price: data.rental ? null : data.price,
    rental: data.rental ? data.price : null,
    negotiable: data.negotiable,
    state: data.state,
    status: data.status,
    categoryId: data.category.uuid,
    tags: data.tags,
    lat: data.location?.lat,
    lng: data.location?.lng,
    images: signedUrls.map((url) => url.public_url),
  };

  return fetch(`${API_URL}/listing`, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      Authorization: token,
    },
    body: JSON.stringify(body),
  });
}
```

фиг. 3.2.54. – Помощна функция за извършване на заявка за качване на обява

Процесът на функцията за създаване на нова обява е почти идентичен с тази за създаване на място, но проверките, параметрите и помощните функции са за качване на обяви.

```
export const handleCreateListing = async (
  data: CreateListingForm,
  images: ImageInputProps[],
  handleError: (newErrors: Partial<CreateProductErrors>) => void,
  t: (key: string) => string,
) => {
  const token = await getAuth('client');
  if (!token) {
    handleError({ global: ['You must be logged in to create a listing'] });
    return;
  }

  const result = createListingSchema.safeParse(data);
  const imagesResult = createImagesSchema.safeParse(images);
  let newErrors: Partial<CreateProductErrors> = {};

  if (!result.success) {
    const formatted = formatErrors(result);
    newErrors = {
      ...newErrors,
      ...extractTranslatedErrors(formatted, t),
    };
  }

  if (!imagesResult.success) {
    const formatted = formatErrors(imagesResult);
    const extracted = extractTranslatedErrors(formatted, t);

    if (
      extracted['root'] &&
      Array.isArray(extracted['root']) &&
      extracted['root'].length
    ) {
      newErrors = {
        ...newErrors,
        images: extracted['root'],
      };
    }
  }
  ...
```

фиг. 3.2.55. - Първа част от функцията за качване на обява – валидиране

```
const signedUrls = await getSignedUrls(images, data.status, token);
  const uploadResults = await uploadImages(signedUrls, images);
  const finalErrors = uploadResults
    .filter((res) => !res)
    .map(() => 'Error uploading image');

  if (finalErrors.length) {
    handleError({ images: finalErrors });
    return;
  }

  const createListingResponse = await createListing(data, signedUrls, token);
  if (!createListingResponse.ok) {
    const listing = (await createListingResponse.json()) as {
      message: string;
    };
    handleError({ global: [listing.message] });
    return;
  }

  return (await createListingResponse.json()) as FullListingResponse;
};
```

фиг. 3.2.56. – Втора част от функцията за качване на обява

3.2.14. Качване на коментар и ревю

За разлика от предишните две функции за създаване на клиентско съдържание, тези две са сравнително по-прости и затова се изпълняват на сървъра, което ги прави сървърни функции.

Във функциите се спазва следния процес на изпълнение, първо се иззема съдържанието от формата, след което се проверява дали е празно и се връща грешка, ако е. След това се премахват всички празни пространства в началото и края и се проверява със Zod функцията. При възникване на грешки – те се връщат като съобщения на потребителя. След цялото това валидиране на съдържание се прави проверка дали потребителят е регистриран, ако не е – връща се грешка.

```
export default async function postComment(
  listing_uuid: string,
  prevState: unknown,
  formData: FormData,
): Promise<
  | true
  | {
      messages: string[];
    }
> {
  noStore();

  const content = formData.get('comment') as string;

  if (!content || !listing_uuid) {
    return {
      messages: ['errors.empty'],
    };
  }

  const newContent = content.trim();

  const validatedData = CreateCommentSchema.safeParse({
    content: newContent,
  });

  if (!validatedData.success) {
    return {
      messages: validatedData.error.issues.map((issue) => issue.message),
    };
  }

  const auth = await getAuth('client');

  if (!auth) {
    return {
      messages: ['errors.unauthorized'],
    };
  }

  ...
}
```

фиг. 3.2.57. – Първа част от функцията за качване на коментар на обява – валидиране на данни

След това се прави опит за качване на коментара или ревюто, като при успешно публикуване, се ревалидират таговете за кеширане на коментарите / ревютата под дадения пост и функцията връща истина.

```
try {
  const res = await fetch(`${API_URL}/listing/${listing_uuid}/comment`, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      Authorization: auth,
    },
    body: JSON.stringify({
      content: newContent,
      listing_uuid,
    }),
  });

  if (res.status === 201) {
    revalidateTag(`/listing/${listing_uuid}/comment`);
    return true;
  } else {
    const extractedServerErrors: ExtractedServerErrors =
      await extractServerErrors(res);

    if (extractedServerErrors) {
      return {
        messages: Object.values(extractedServerErrors.errors).flat(),
      };
    }

    return {
      messages: ['errors.500'],
    };
  }
} catch (err) {
  console.error('COMMENT ERRORS', err);
  return {
    messages: ['errors.500'],
  };
}
```

фиг. 3.2.58. – Продължение на функцията за качване на коментар на обява – създаване и изпълнение на заявка към ППИ

Качването на ревю е подобно на качването на коментар и може да бъде разгледано във файла „/src/actions/review.ts” на електронния носител или в GitHub хранилището.

3.2.15. Качване на оценка

Друга проста функция за качване на потребителско съдържание е тази за оценяване на потребители. Тя също е сървърна, но не използва HTML форми за нейното изпълнение. Тя приема потребителско име,

стойност и таг за ревалидиране (на даден потребител). Първата проверка, която се извършва, е дали потребителят е регистриран, при успешно преминаване се извършва POST заявката за оценяване на друг потребител.

```
export const rate = async (
  username: string,
  value: number,
  revalidate?: string,
) => {
  const auth = await getAuth('client');
  if (!auth) {
    return;
  }

  await fetch(`${API_URL}/user/rate/${username}`, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      Authorization: auth,
    },
    body: JSON.stringify({
      rating: value,
    }),
  })
    .then((r) => r.json())
    .then((data) => {
      console.log(data);
      revalidate && revalidateTag(revalidate);
      revalidateTag('users');
    });
};
```

фиг. 3.2.59. – Сървърна функция за публикуване на оценка

3.2.16. Търсене – глобално и частично

Функцията за търсене приема един задължителен аргумент – самото търсене, като разполага и с 3 по желание – ИН на категория, номер на страница и лимит. В зависимост дали категорията е подадена, се изпълняват две различни заявки. При неподадена категория се извлича кратка информация за глобалното търсене. Ако е подадена, се извлича спрямо категория, страница и лимит на страница с помощта на *paginatedFetch()* функцията. Функцията за търсене може да бъде разгледана на *фиг. 3.2.60.*, която се намира на следващата страница.

```
export const getSearch = async (
  search: string,
  category?: string,
  page = 1,
  limit = 12,
) => {
  // === GLOBAL SEARCH ===
  if (category === undefined) {
    const res = await fetch(`${API_URL}/search?search=${search}`, {
      next: {
        revalidate: 60,
      },
    });

    return (await res.json()) as GlobalSearchResult;
  }

  // === CATEGORY SEARCH ===
  return paginatedFetch<SearchType>(
    `/search?search=${search}&category=${category}`,
    page,
    limit,
    {
      next: {
        revalidate: 60,
      },
      cache: 'no-store',
    },
  );
};
```

фиг. 3.2.60. – Функция за извличане на търсене

В компонента (част от него показана на *фиг 3.2.61.*) за търсене се извличат данните, като се подават на *getSearch()* функцията – търсене, категория, страница и лимит, от които всички, без search, може да са *undefined*. Ако категорията не е посочена, това означава, че търсенето е глобално, ако е – се извличат отговор със разделение по страници, както и общ брой на елементи.

```
const data = await getSearch(search, category, page, limit);

  // === GLOBAL SEARCH ===

  if (category === undefined) {
    return (
      <SearchWrapper>
        <GlobalSearch data={data as GlobalSearchResult} />
      </SearchWrapper>
    );
  }

  // === CATEGORY SEARCH ===

  const { data: categoryData } = data as PaginatedResponse<
    ShortListingResponse | CardPlace | User
  >;

  const { totalCount } = data as PaginatedResponse<
    ShortListingResponse | CardPlace | User
  >;
```

фиг. 3.2.61. – Извличане на резултат от търсене и разпределение спрямо вида

3.2.17. Извличане на публичен или личен профил

Създадени са две отделни функции, които се свързват с различни пътища на приложно-програмния интерфейс за извличане на данни на потребител. Първата функция *getUser()* е за извличане на данни, до които имат достъп всички – публичен профил на друг потребител. Функцията се извиква с подадено потребителско име и без изискването на каквато и да е авторизация.

```
const getUser = async (username: string) => {
  const response = await fetch(`${API_URL}/user/${username}`, {
    next: {
      revalidate: 1,
      tags: [`/user/${username}`],
    },
  });

  if (!response.ok) return null;

  return (await response.json()) as PublicProfile;
};
```

фиг. 3.2.62. – Извличане на информация за публичен профил на потребител

Втората функция *getProfile()* единствено се нуждае от токен за достъп. Спрямо него приложно-програмния интерфейс разбира кой е потребителя и връща както негова публична информация, така и негова защитена / частна такава.

```
const getProfile = async (auth: string) => {
  const res = await fetch(`${API_URL}/user/me`, {
    headers: {
      Authorization: auth,
    },
    next: {
      tags: ['user'],
    },
  });

  if (res.status === 401 || res.status === 403) {
    return null;
  }

  return (await res.json()) as UserProfile | null;
};
```

фиг. 3.2.63. - Извличане на информация за собствен профил на потребител

Обаче информацията, която е нужна за визуализирането на потребителските профили не е достатъчна, тъй като това извлича само потребителските данни, без неговите публикации – обяви и места. Затова примерна употреба може да се разгледа на фиг. 3.2.64., където една от горепосочените функции се използва в съчетание с функциите за извличане на обяви и места за даден потребител.

```
const auth = await getAuth('ssr');
if (!auth) notFound();

const profile = await getProfile(auth);
const listingsData = await getListings(1, 6, 'me', undefined, auth);
const placesData = await getPlaces(1, 4, 'me', undefined, auth);

if (!profile || !listingsData || !placesData) notFound();
```

фиг. 3.2.64. – Примерна употреба на функциите за извличане на данни – част от извличането в профилна страница на регистриран потребител

Глава IV

Ръководство на потребителя

4.1. Инсталация

Предварително инсталиран софтуер може да включва: git, npm, pnpm, node и други.

Git:

- За инсталиране на git на Windows: https://git-scm.com/
- За инсталиране на git на macOS:

>>> xcode-select --install

- За инсталиране на git на GNU/Linux:

>>> [общ пример] {package_mgr} {install_cmd} git

>>> [debian/ubuntu] sudo apt install git

>>> [arch] sudo pacman -S git - y

NVM (Node Version Manager) - за инсталиране на различни версии на node.js и тяхното менажиране (проектът използва v20.10.0):

- За инсталиране на nvm на Windows:

https://github.com/coreybutler/nvm-windows/releases

- За инсталиране на nvm на macOS:

>>> brew install nvm

- За инсталиране на nvm на GNU/Linux:

>>> curl https://raw.githubusercontent.com/creationix/nvm /v0.39.7/install.sh | bash

>>> source ~/.bashrc

Инсталиране на NodeJS и NPM с NVM:

>>> nvm install 20.10.0

Инсталиране на PNPM:

- Windows:

>>> iwr https://get.pnpm.io/install.ps1 -useb | iex

- POSIX (macOS и GNU/Linux):

>>> curl -fsSL https://get.pnpm.io/install.sh | sh -

Инсталиране на Docker:

- Windows: *https://docs.docker.com/desktop/install/windows-install/*
- macOS: *https://docs.docker.com/desktop/install/mac-install/*
- GNU/Linux: *https://docs.docker.com/engine/install/*

4.1.1. Инсталация на базите данни

За предпочитане базите данни (PostgreSQL и Redis) се изпълняват в Docker контейнери. За тяхното инсталиране трябва да се следват следните инструкции:

Redis:

Създаване на Docker Volume за съхранение на данни – "redis-data".

```
docker volume create redis-data
```

фиг. 4.1. – Създаване на Docker Volume

След това трябва да се създаде контейнер, чиито параметри посочват порт, на който се използва, наименование на контейнера, какъв

image от Docker Hub да използва, както и да се пояснят пароли за default потребителя на базата данни.

```
docker run -d \
  -h redis \
  -e REDIS_PASSWORD=redis \
  -v redis-data:/data \
  -p 6379:6379 \
  --name redis \
  --restart always \
  redis:5.0.5-alpine3.9 /bin/sh -c 'redis-server --appendonly yes --requirepass ${REDIS_PASSWORD}'
```

фиг. 4.2. – Създаване на контейнер за база данни - Redis

PostgreSQL:

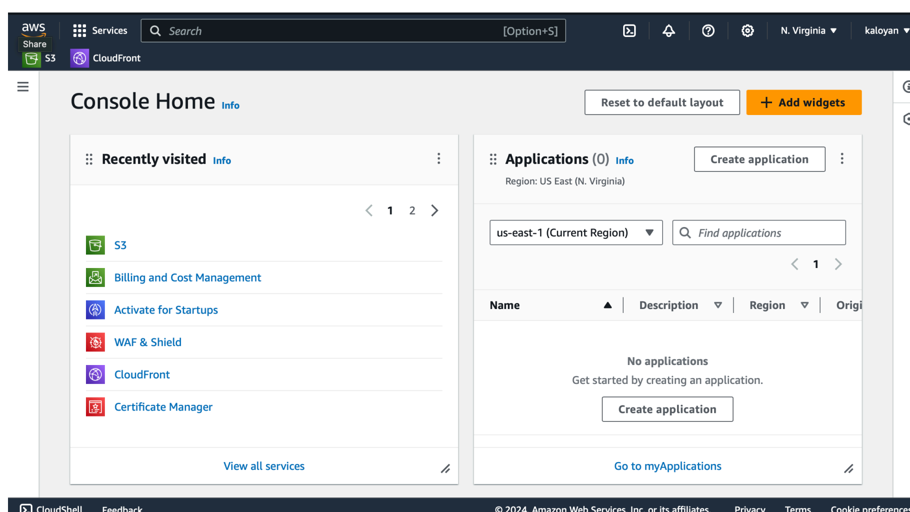
За създаване на контейнер с параметри по подразбиране със следната команда. Като повече параметри може да бъдат разгледани в документацията на страницата на официалния контейнер.

```
docker run --name postgres -e POSTGRES_PASSWORD=mysecretpassword -d postgres
```

фиг. 4.3. – Създаване на контейнер за база данни – PostgreSQL

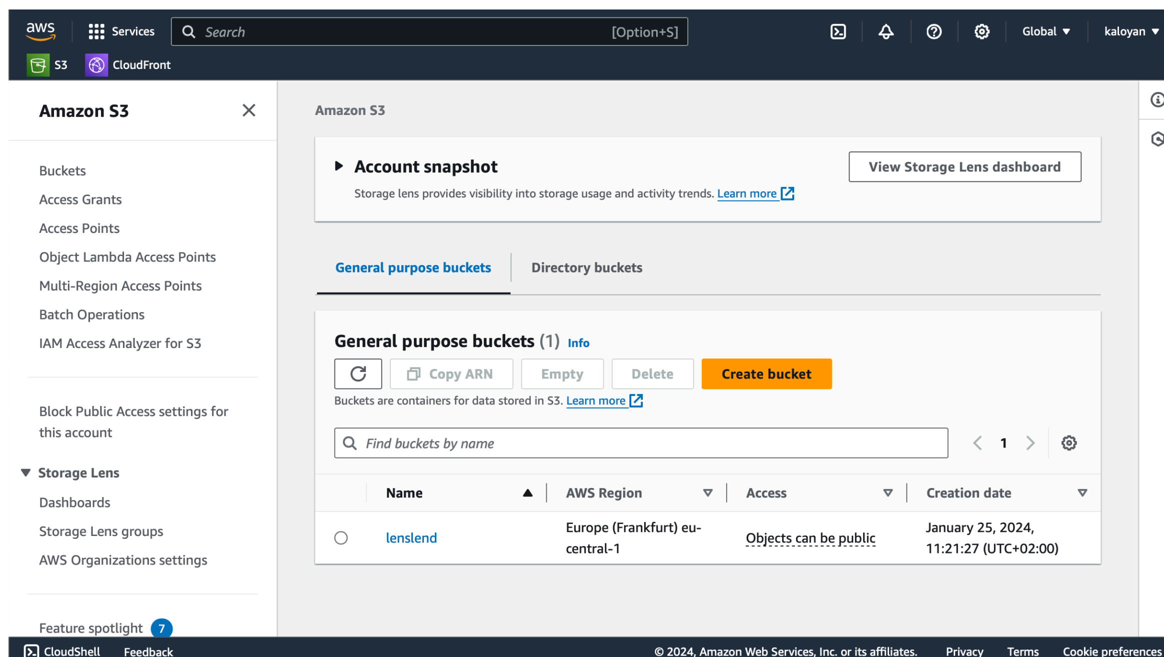
4.1.2. Създаване на Amazon S3 Bucket

Първата стъпка е да се създаде Amazon AWS акаунт на: http://aws.amazon.com/ , и да се обвърже със Вашата банкова сметка. След влизане в AWS конзолата.



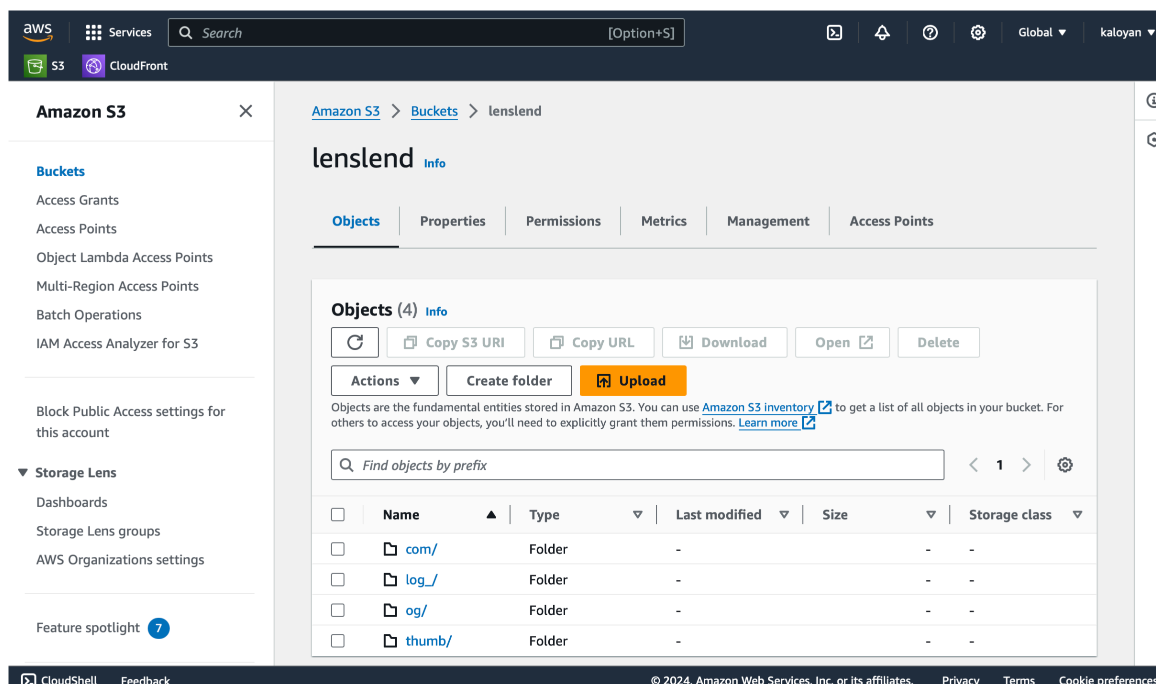
фиг. 4.4. – Интерфейс на AWS Конзола

За създаване на S3 кофа, трябва да се навигира до S3 интерфейса. И там да се избере „Create Bucket” за създаване на нова кофа.



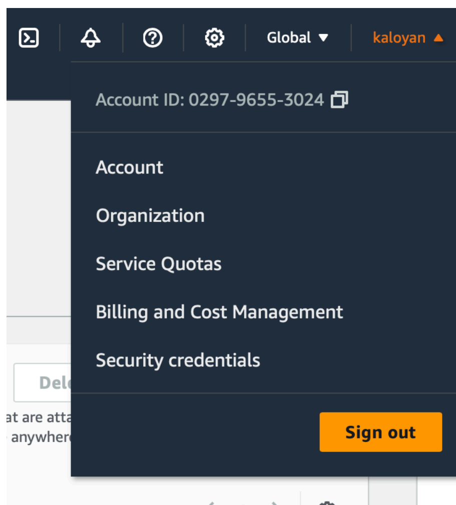
фиг. 4.5. – Amazon S3

След избиране на името и конфигурация по Ваш избор на S3 кофата. Може да разделите кофата на директории. И след това трябва да се надстрои конфигурацията за public-read ACL в Permissions таба.



фиг. 4.6. – Конфигурация на AWS S3 кофа

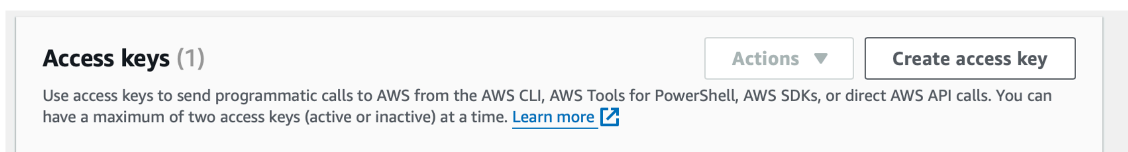
За взимане на информацията за достъп – ключове за достъп. Трябва да се избере потребителското име от навигационното поле и след това да се избере линка „Security credentials".



фиг. 4.7. – Навигационно поле за потребител

След това се създават тайни ключове за достъп с „Create Access Key" – 2 ключа на име:

- AWS_ACCESS_KEY_ID
- AWS_SECRET_ACCESS_KEY

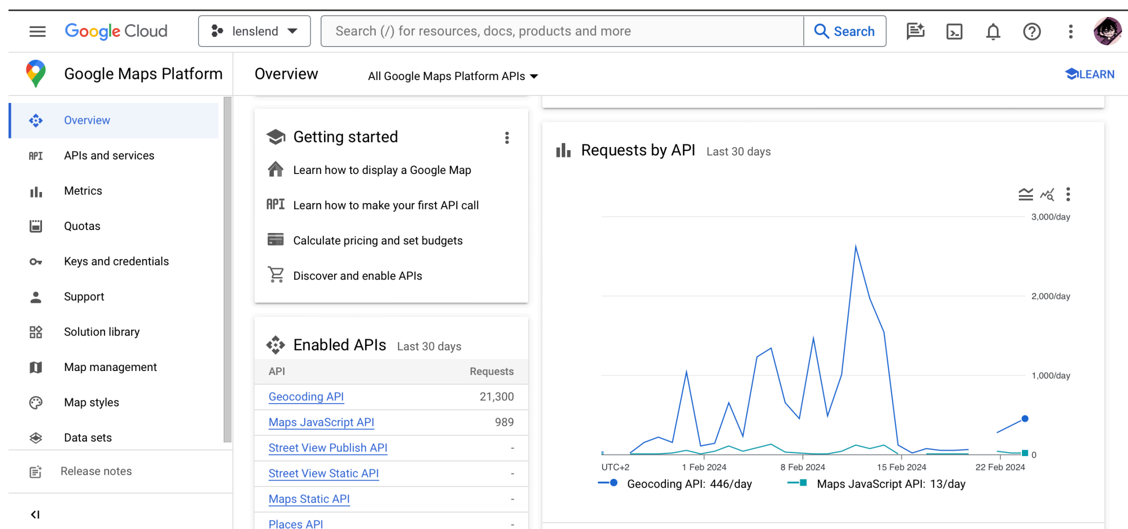


фиг. 4.8. – Създаване на ключ за достъп

4.1.3. Създаване на Google Maps API ключ

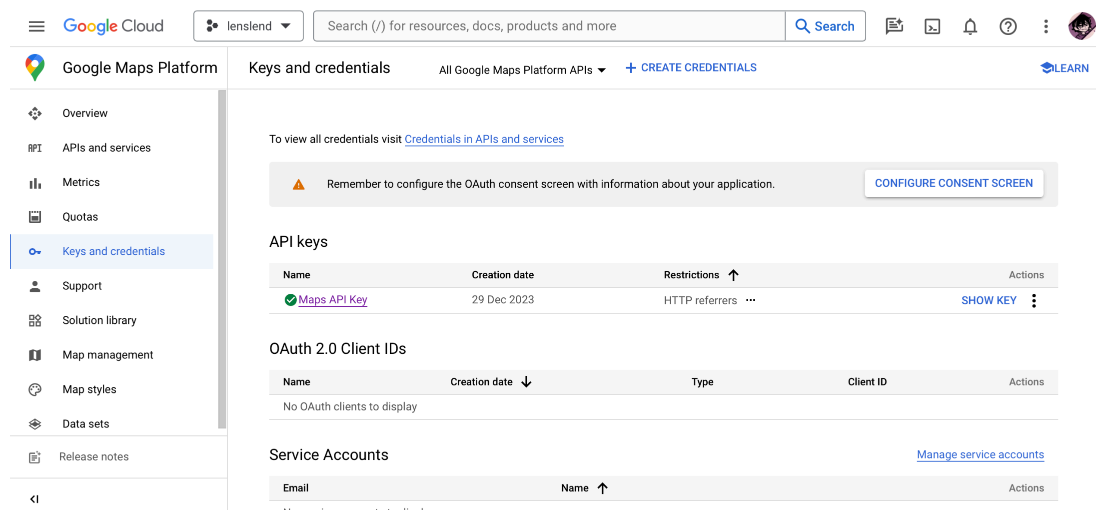
Първата стъпка е да се създаде Google акаунт, като след това трябва да се обвърже банкова сметка с него и тя да бъде потвърдена. След това може да се продължи към платформата на Google за Google Maps – Google Maps Platform. В нея трябва да се създадат две неща – ключ за достъп до

Google Maps приложно-програмния интерфейс и специална Google Maps карта, на която да може да се използва идентификационния номер.



фиг. 4.9. – Начална страница на Google Maps Platform

За създаване на ключ за достъп трябва да се отиде на страница „Keys and credentials" в страничното навигационно поле. Като след това се създава ключ за достъп чрез бутона „CREATE CREDENTIALS".



фиг. 4.10. – Страница „Key and credentials" на Google Maps Platform

Като при неговата конфигурация задължително трябва да бъдат избрани услугите за геолокиране, използване на карти и геокодиране.

Както и с цел сигурност, да бъде защитен за употреба на позволени домейни:

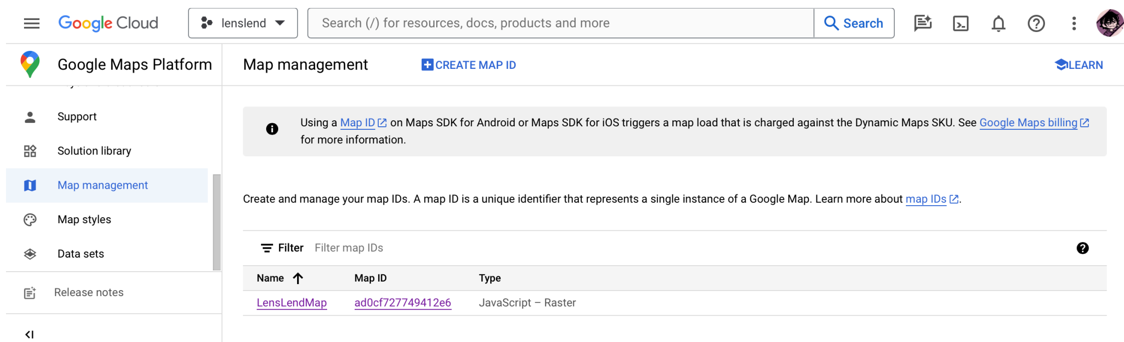
ADD

Filter Enter property name or value

	Status	Website	Edit
		.kaloyan.tech/	
		192.168.1.198:3000/*	
		192.168.1.200:3000/*	
		http://192.168.1.198:3000/*	
		http://192.168.1.200:3000/*	
		https://lenslend-dev.kaloyan.tech	
		https://lenslend-dev.kaloyan.tech/*	
		kaloyan.tech	
		lenslend-frontend.vercel.app/*	
		localhost:3000/*	

фиг. 4.11. – Защита чрез разрешаване само на определени домейни

За създаване на специална карта и извличане на нейното ID, трябва да се отиде на страницата „Map management” и с бутона „CREATE MAP ID”, то да бъде създадено.



фиг. 4.12. – Създаване на Map ID на страница „Map Management”

4.1.4. Инсталация на приложно-програмен интерфейс

Първата стъпка, която трябва да се изпълни е да се клонира (изтегли) GitHub хранилището на приложно-програмния интерфейс:

```
>>> git clone https://github.com/KokosTech/lenslend-backend
>>> cd lenslend-backend
```

>>> pnpm install

Попълване на „.env” конфигурационен файл:

```
DATABASE_URL="postgresql://<ИМЕ_БАЗА_ДАННИ>:<ПОТРЕБИТЕЛСКО_ИМЕ>
@<ДОМЕЙН_ИЛИ_IP_АДРЕС>:<ПОРТ>/<ИМЕ_БАЗА_ДАННИ>
?schema=public&pool_timeout=15&connection_limit=5"
JWT_SECRET="<КЛЮЧ_ЗА_КОДИРАНЕ_НА_JWT_AUTH>"
JWT_EXPIRES_IN="1d"
JWT_REFRESH_SECRET="<КЛЮЧ_ЗА_КОДИРАНЕ_НА_JWT_REFRESH>"
JWT_REFRESH_EXPIRES_IN="30d"

# REDIS
REDIS_URL="redis://<ПОТРЕБИТЕЛСКО_ИМЕ>:<ПАРОЛА>@<ДОМЕЙН_ИЛИ_IP_АДРЕС>:<ПОРТ>"
REDIS_USERNAME="<ПОТРЕБИТЕЛСКО_ИМЕ>"
REDIS_PASSWORD="<ПАРОЛА>"
REDIS_NAME="<ИМЕ>"
REDIS_DATABASE="0"

# AWS
AWS_S3_REGION="<РЕГИОН_S3>"
AWS_ACCESS_KEY_ID="<ID_КЛЮЧ_ЗА_ДОСТЪП_S3>"
AWS_SECRET_ACCESS_KEY="<КЛЮЧ_ЗА_ДОСТЪП_S3>"

AWS_S3_BUCKET_NAME="<ИМЕ_КУТИЯ>"
AWS_S3_FOLDER="<ИМЕ_ДИРЕКТОРИЯ>"

AWS_CLOUDFRONT_URL="<URL_ЗА_CLOUDFRONT"

NODE_ENV="development"
VERSION="0.0.1"
```

фиг. 4.13. – Конфигурационен “.env” за клиентски софтуер

>>> pnpm start:dev

4.1.5. Инсталация на клиентската част

Първата стъпка, която трябва да се изпълни е да се клонира (изтегли) GitHub хранилището на клиентската част:

>>> git clone https://github.com/KokosTech/lenslend-frontend

>>> cd lenslend-frontend

>>> pnpm install

Попълване на „.env” конфигурационен файл:

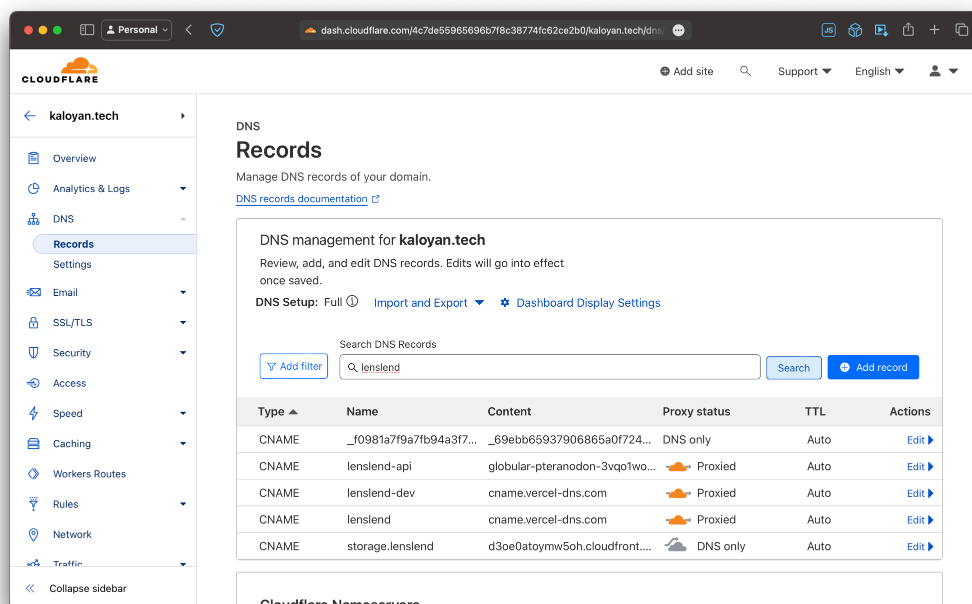
```
NEXT_PUBLIC_GOOGLE_MAPS_API_KEY=<КЛЮЧ_ЗА_GOOGLE_MAPS_API>
NEXT_PUBLIC_GOOGLE_MAPS_ID=<ID_ЗА_GOOGLE_MAPS_ID>
NEXT_PUBLIC_API_URL=https://lenslend-api.kaloyan.tech
NEXT_PUBLIC_URL=https://lenslend.kaloyan.tech
```

фиг. 4.14. – Конфигурационен “.env” за клиентски софтуер

>>> pnpm dev

4.1.6. Вдигане на софтуера в облака

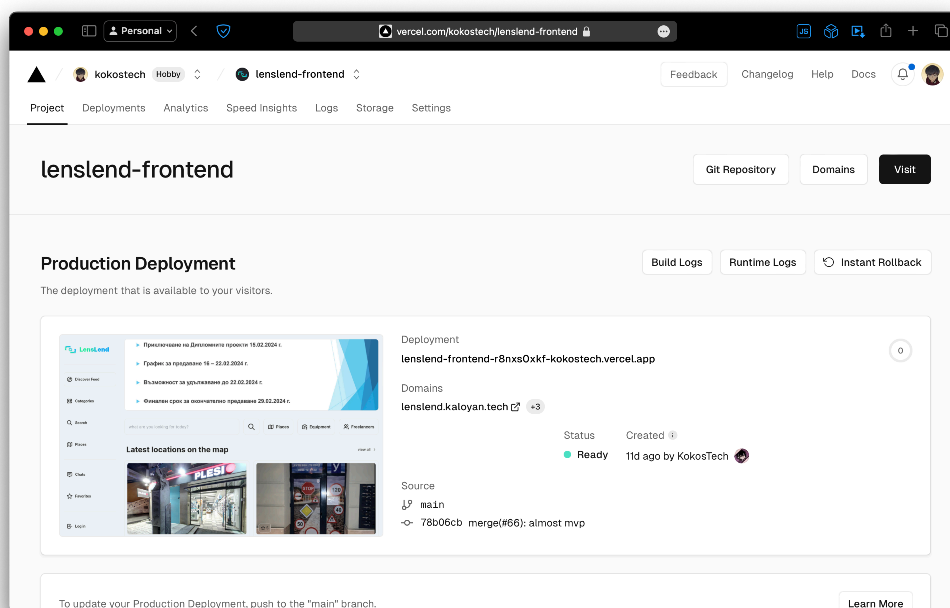
За менажирането на домейни и събдомейни се използва платформата Cloudflare.



фиг. 4.15. – Cloudflare на LensLend

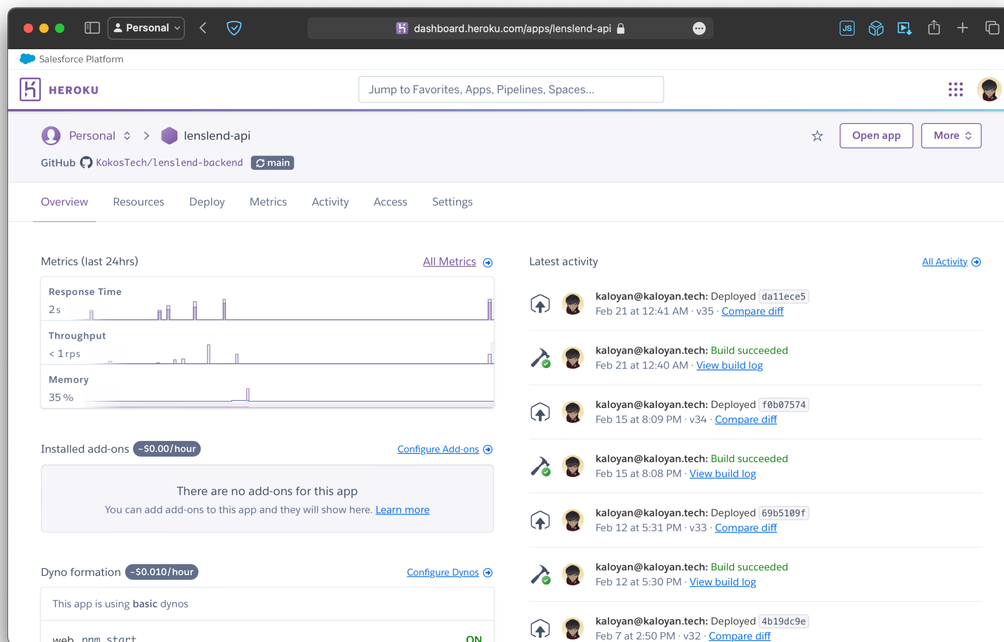
Платформата LensLend е вдигната на 3 места:

- Vercel – за клиентския софтуер: https://lenslend.kaloyan.tech



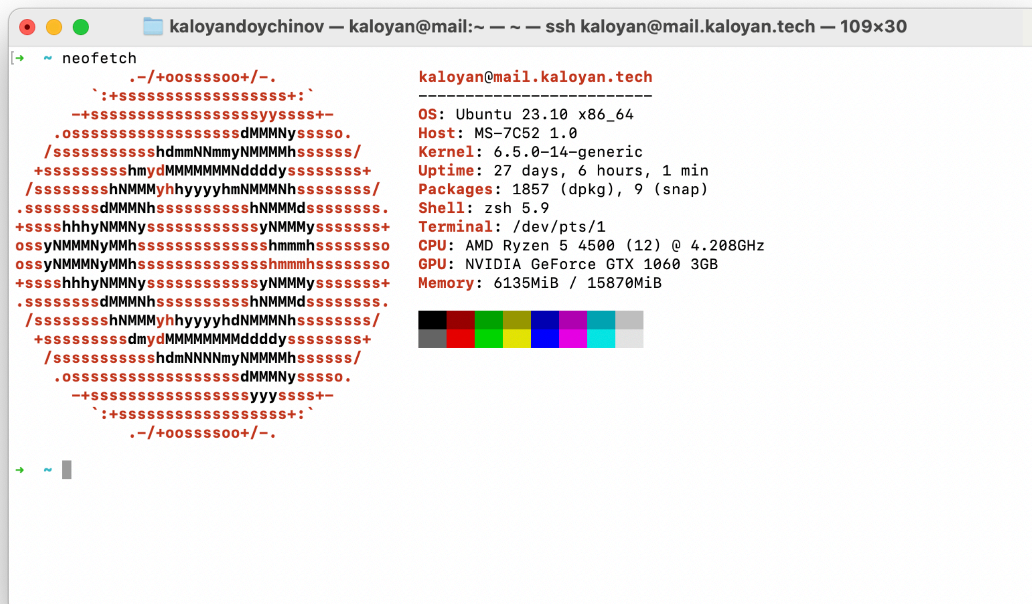
фиг. 4.16. – Vercel на LensLend

- Heroku – за сървърния софтуер: https://lenslend-api.kaloyan.tech



фиг. 4.17. – Heroku на LensLend

- Домашен сървър на GNU/Linux с Docker за изпълнението на базите данни: https://mail.kaloyan.tech



фиг. 4.18 – Информация за „домашния“ GNU/Linux сървър

kaloyandoychinov — kaloyan@mail:~ — ~ — ssh kaloyan@mail.kaloyan.tech — 109×30

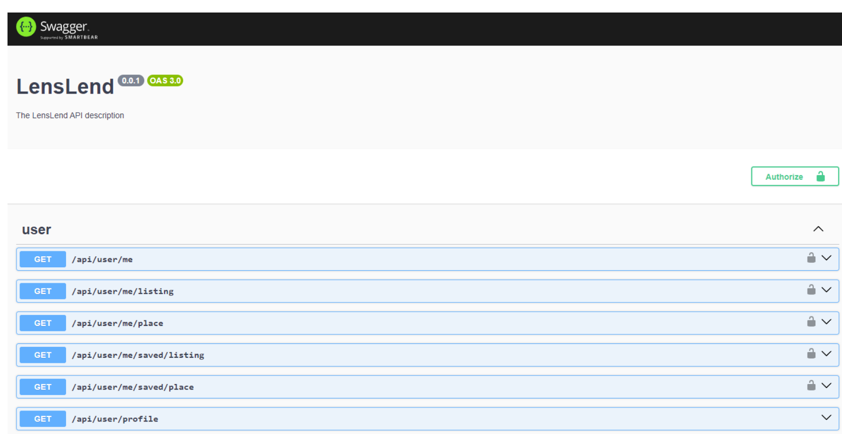
```
➜  ~ docker ps | grep 'redis'
e5df04bdc89e   redis/redis-stack-server:latest      "/entrypoint.sh"        8 weeks ago     Up 3 weeks
       0.0.0.0:6380->6379/tcp, :::6380->6379/tcp
                                                                              redis-stack-server
➜  ~ psql --version
psql (PostgreSQL) 15.5 (Ubuntu 15.5-0ubuntu0.23.10.1)
➜  ~
```

фиг. 4.19. – Информация за Redis и PostgreSQL на GNU/Linux сървъра

4.2. Инструкции за използване на системата

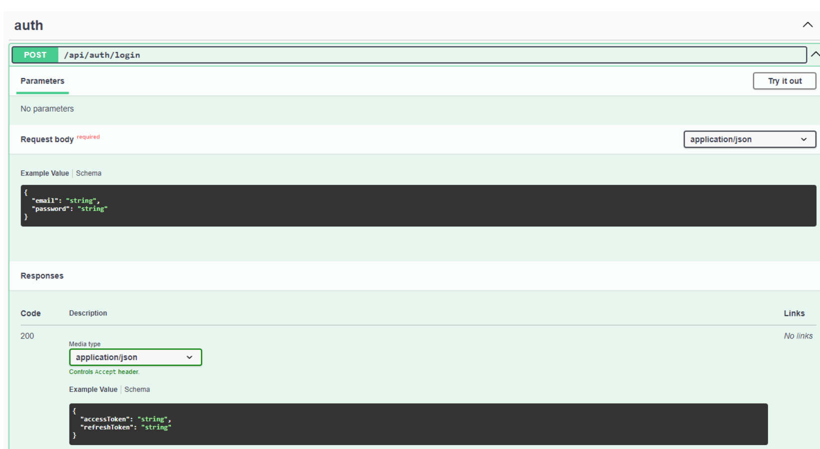
4.2.1. Административна употреба в среда на разработчика – Swagger / OpenAPI документация

След успешно стартиране на приложно-програмния интерфейс, административните органи или разработчици могат да достъпят онлайн документацията на ППИ на "http(s)://{ADDRESS}/api". В тази документация на Open-API (Swagger) могат да разглеждат различните пътища на интерфейса, какви са им входните и изходните данни, както и да правят различни заявки с предварително попълнена информация.



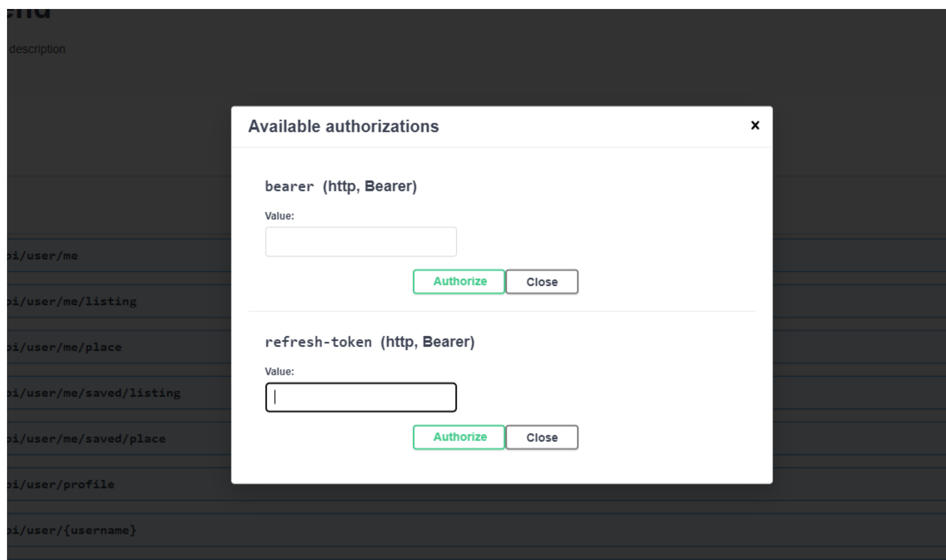
фиг. 4.20. - Изглед на онлайн документацията Open-API

Потребителят може да се регистрира или влезе също чрез подаване на заявки.



фиг. 4.21. - Отворена заявка и показани примери за входни и изходни данни, заедно с бутон за пускане на заявка

След което трябва да си въведе получените "access" и "refresh" токени на обозначените места (или главно място) с катинарче, за да потребителят да има достъп до защитени пътища или пътища, които изискват (дори опционална) авторизация.



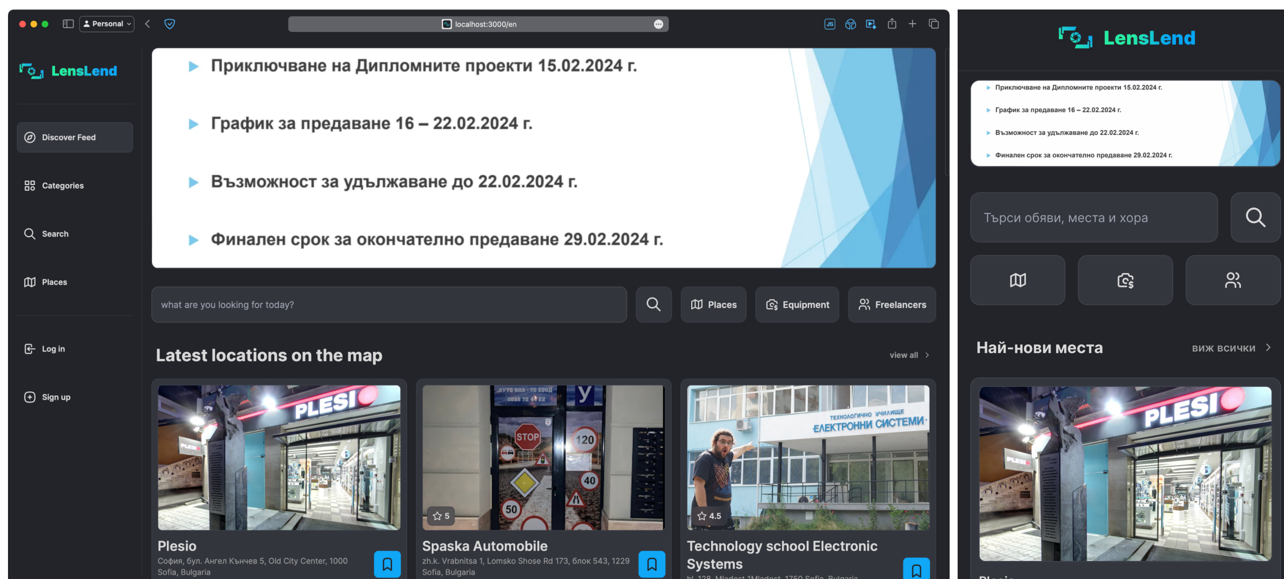
фиг. 4.22. - Въвеждане на Authorization токени

4.2.2. Клиентска употреба

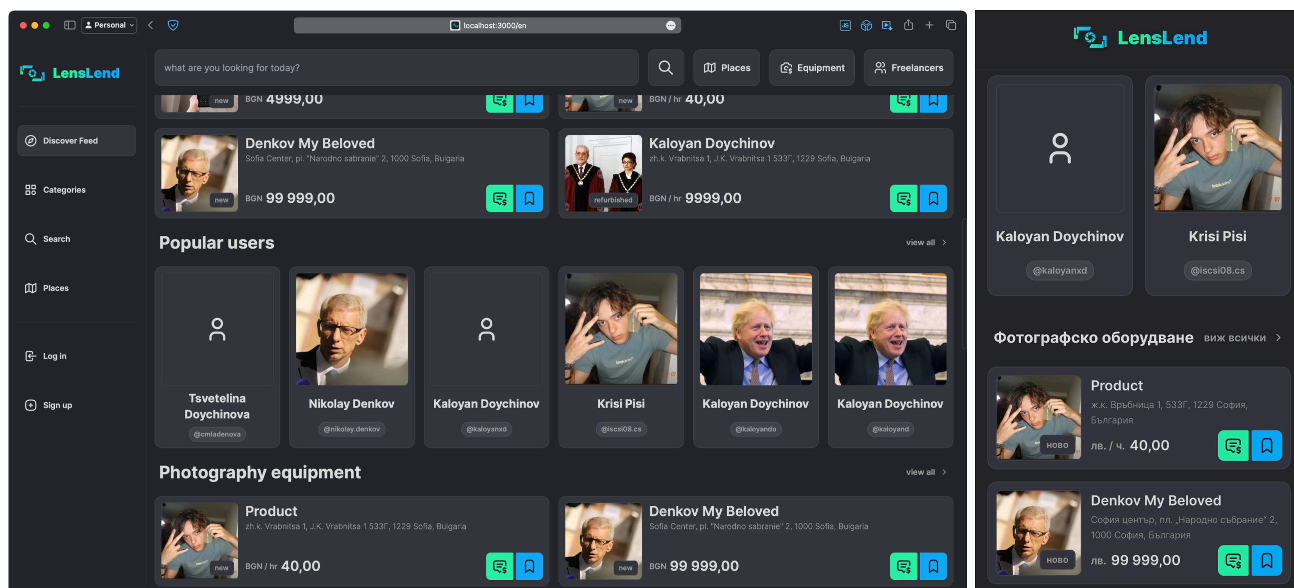
Началната страница е първата страница, която потребителя може да разгледа без дори да влиза в профила си. Идеята на приложението е да бъде достъпно дори и без създаване или влизане от потребителски профил. Това се отнася за най-простите му функционалности, като разглеждане на началната страница, категории, търсене, използване на карти и гледане на обяви и места. Потребителят трябва да е в акаунт, за да може да оценява, коментира, оставя ревюта или запазва места и обяви.

В началната страница може да се разгледат най-новите места, обяви, потребители и т.н. Освен това на нея има интерфейс за търсене на потребители, обяви и места, както и голям банер, който се за вътрешни реклами и обявления в приложения. За приложението има глобално

навигационно меню, което се показва на почти всички страници с изключение на няколко.



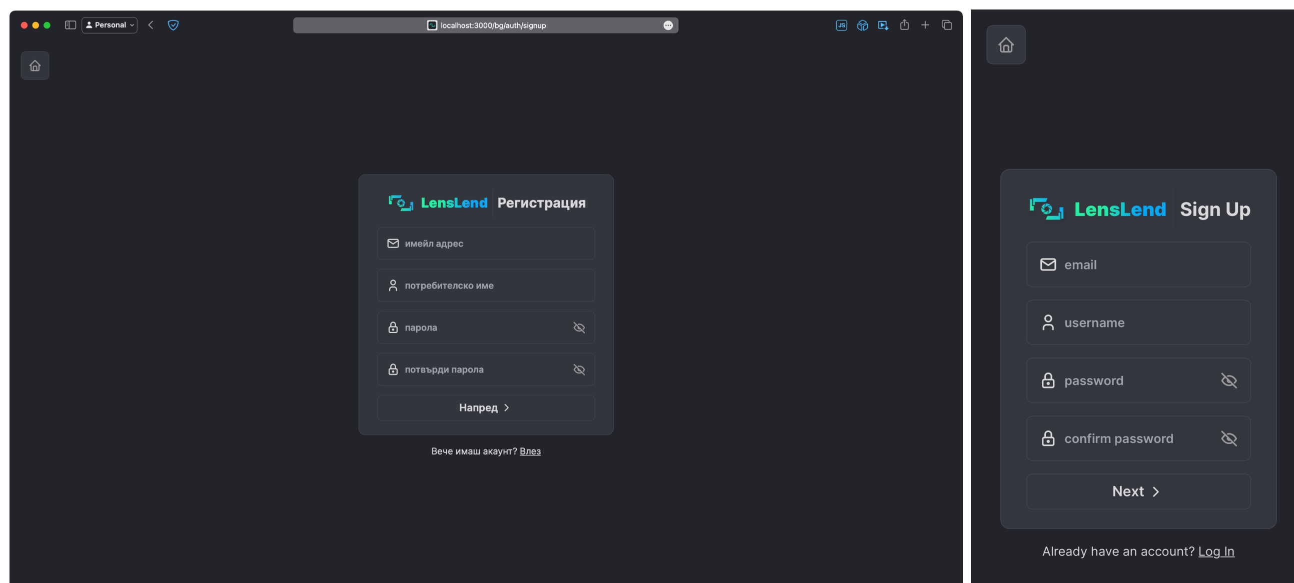
фиг. 4.23. – Начален екран част 1-ва



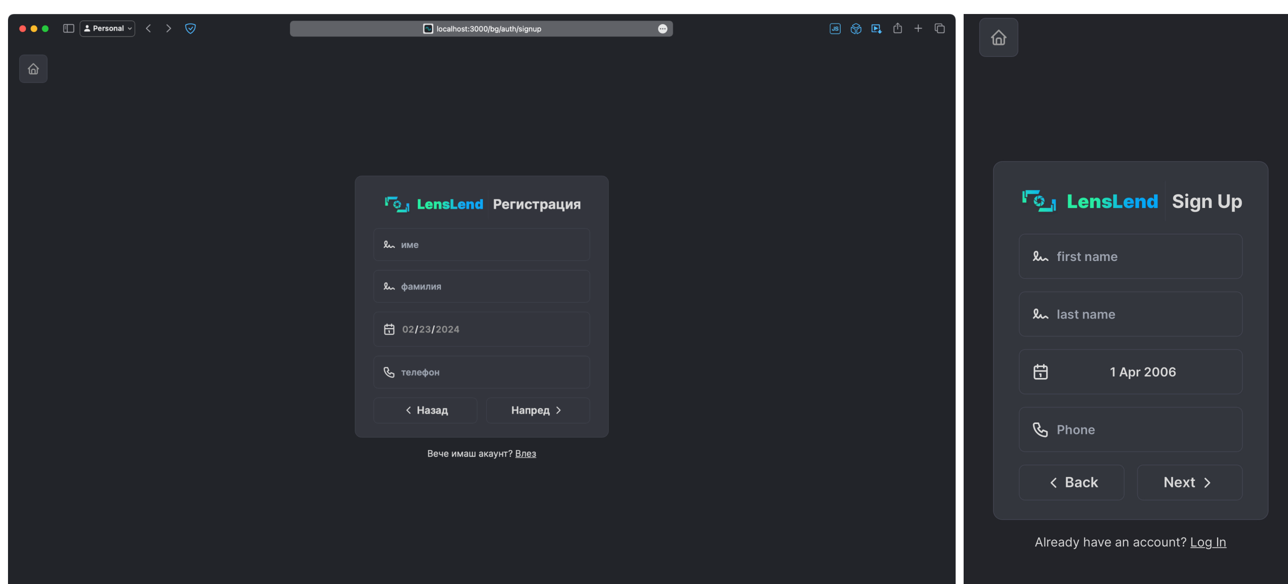
фиг. 4.24. – Начален екран част 2-ра

Ако потребителят желае да се регистрира може да отиде на регистрационната страница, като използва навигационното поле. Регистрацията е разделена на четири части – информация за акаунта, информация за профила, запознаване с условията за използване на

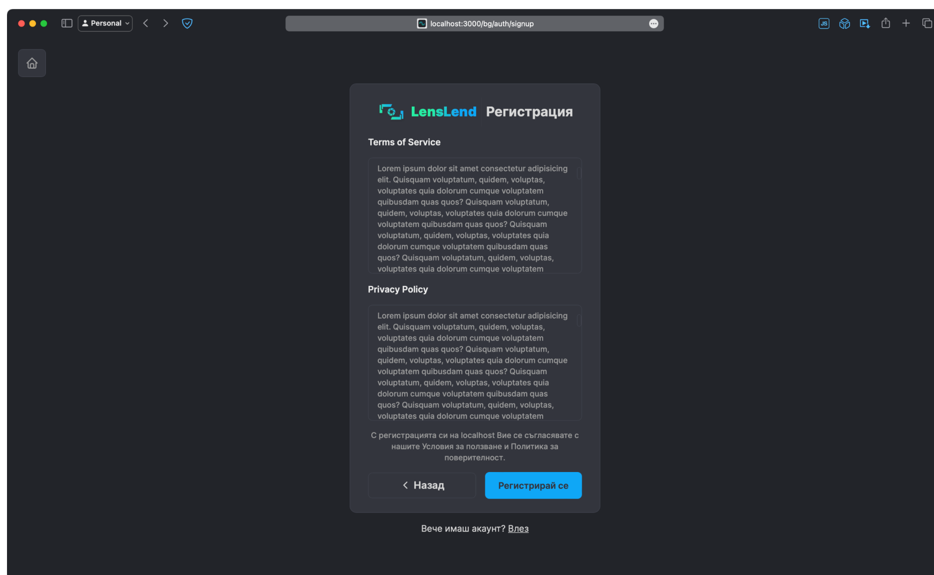
приложението и накрая потвърждаване на имейл адрес и телефонен номер.



фиг. 4.25. – Първа стъпка от процеса за регистрация

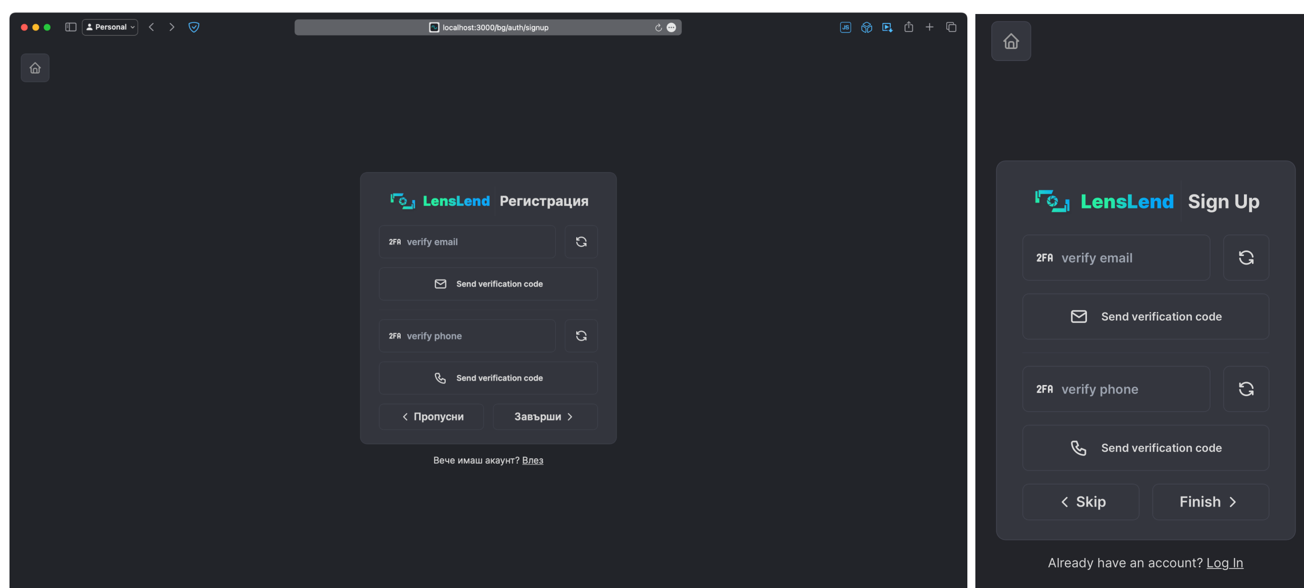


фиг. 4.26. – Втора стъпка от процеса за регистрация



фиг. 4.27. – Трета стъпка от процеса по регистрация

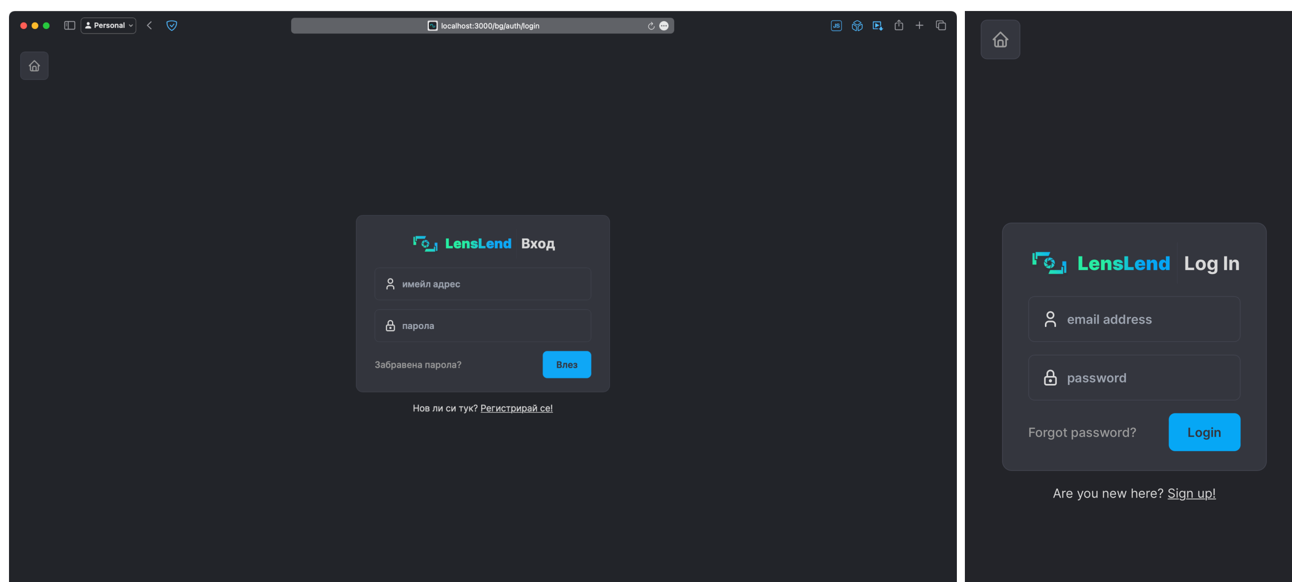
След успешна регистрация идва и време за четвърта стъпка, която е незадължителна – потвърждаване на имейл и телефонен номер. След успешна регистрация, потребителят също така е влязъл в новия си акаунт.



фиг. 4.28. – Четвърта стъпка от процеса по регистрация.

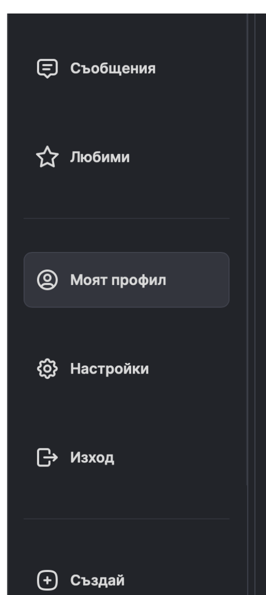
Както може да се забележи на всяка страница в процеса на регистрация, ако потребителят вече има регистрация, той може да влезне с вече съществуващия си акаунт. Такъв бутон също може да бъде

открит в навигационното поле на уеб приложението. След успешен вход, потребителят бива автоматично прехвърлен на началната страница на приложението.



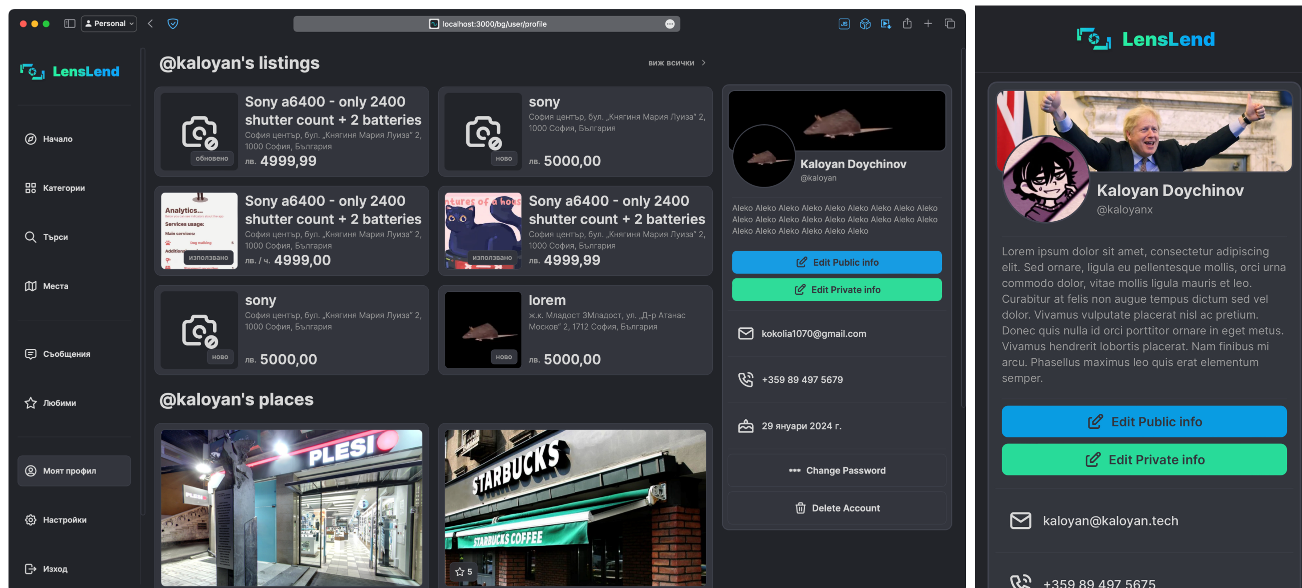
фиг. 4.29. – Екран за вход в потребителски профил

След успешен вход, потребителският интерфейс се позапълва. Навигационното поле има много повече опции – за чат, за любими, за разглеждане на собствен профил, за настройки, за излизане и за създаване на съдържание в приложението.



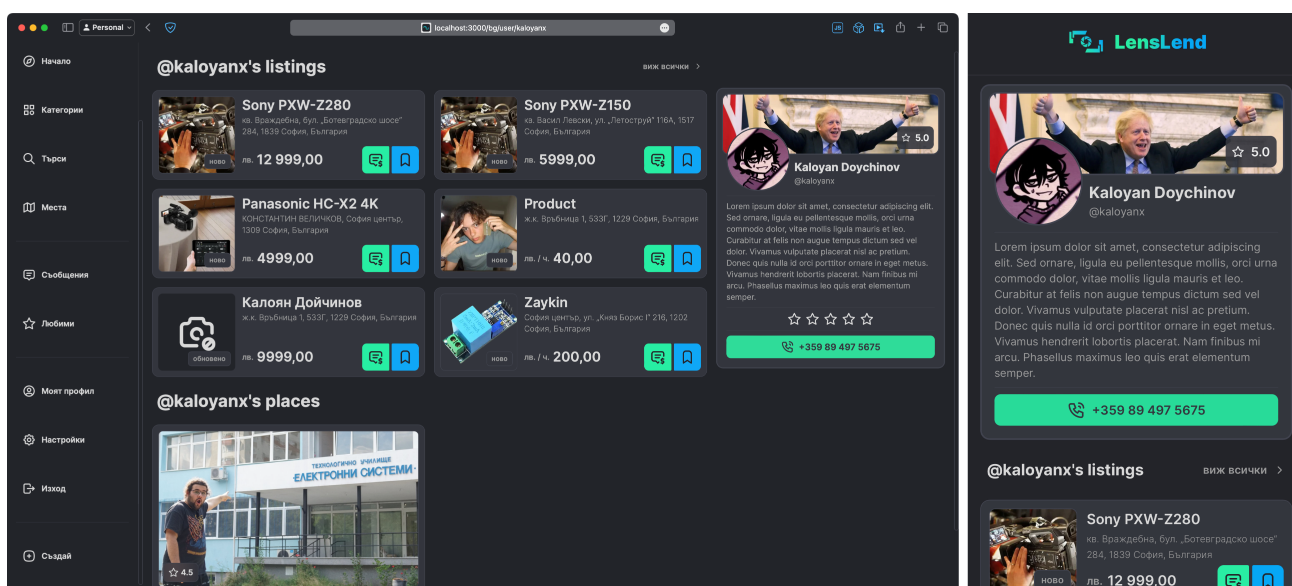
фиг. 4.30. – Навигационни компоненти, достъпни само за влезли потребители

Първата страница, която ще бъде разгледана е профилната страница. В нея се съдържа списък от всички обяви и места на потребителя, както и с неговата частна и публична информация.



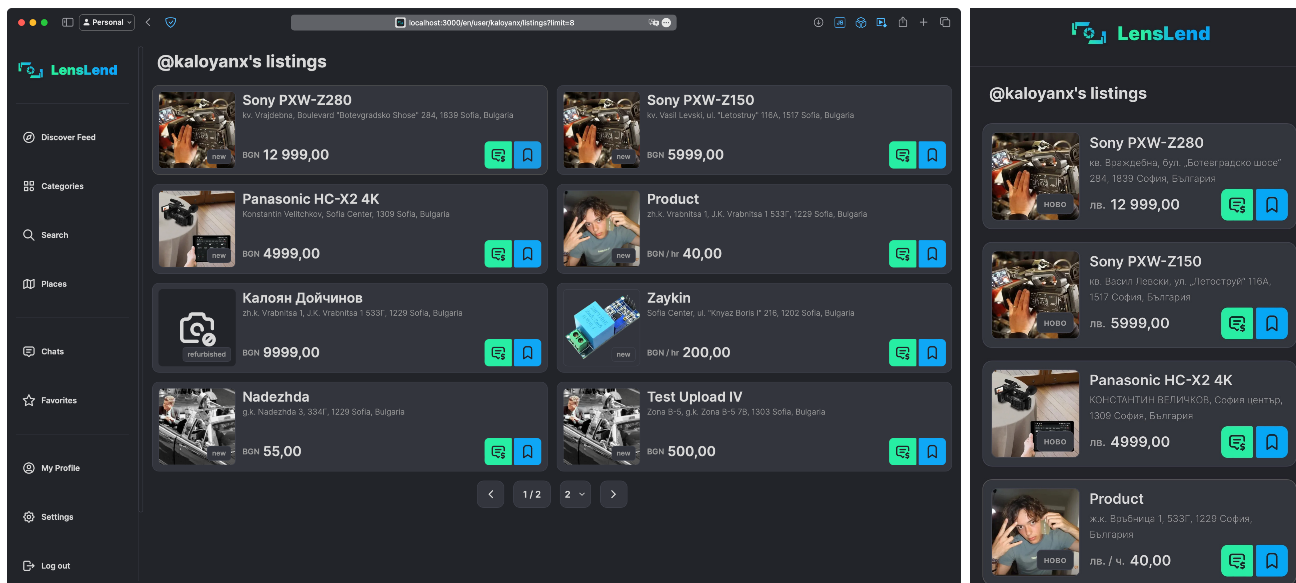
фиг. 4.31. – Профилна страница на регистрирания потребител

Друга подобна страница съществува в приложението за разглеждане на профилна страница, но за чужд потребител. На нея може да се преглеждат само публичните данни, но освен това, може да се оценяват потребителите.



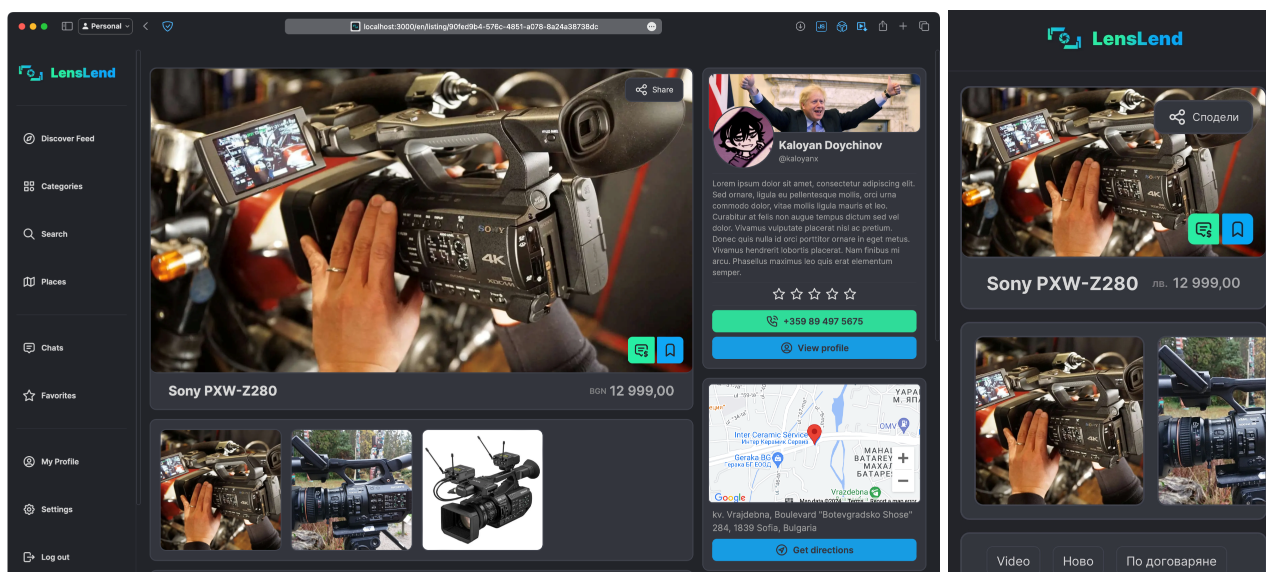
фиг. 4.32. – Публична профилна страница на друг потребител

Ако има повече от 6 обяви или места, ще излезе бутон – „вижте всички“ / „view all”, който завежда потребителя на страница с всички публикации на дадения потребител.

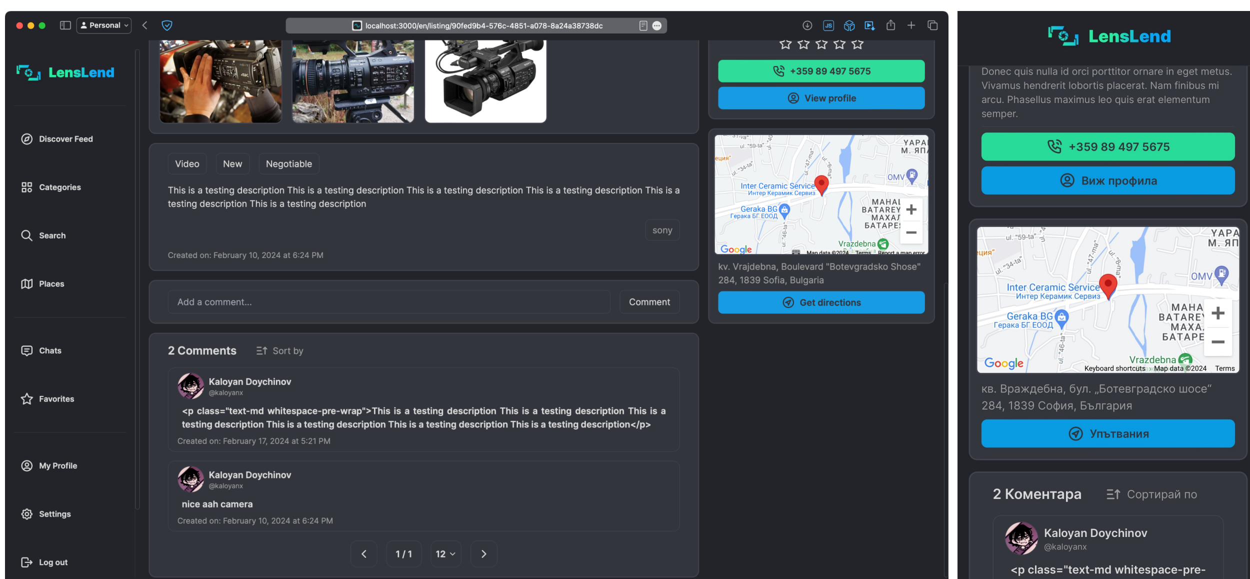


фиг. 4.33. – Обяви на даден потребител

Говорейки за обяви, екранът за обяви съдържа главна снимки, галерия от всички снимки, описание с тагове и времева информация, информация за потребителя и възможност за оценяване, локация, коментари и форма за публикуване на коментар.

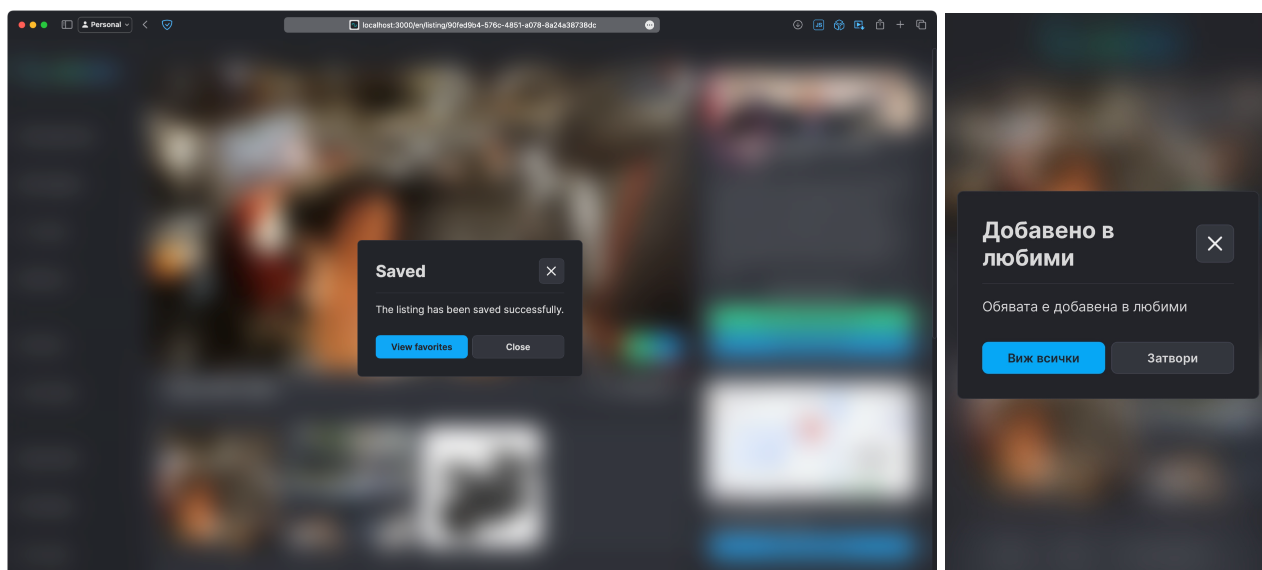


фиг. 4.34. – Част първа от страницата на обява



фиг. 4.35. – Част втора от страницата на обява

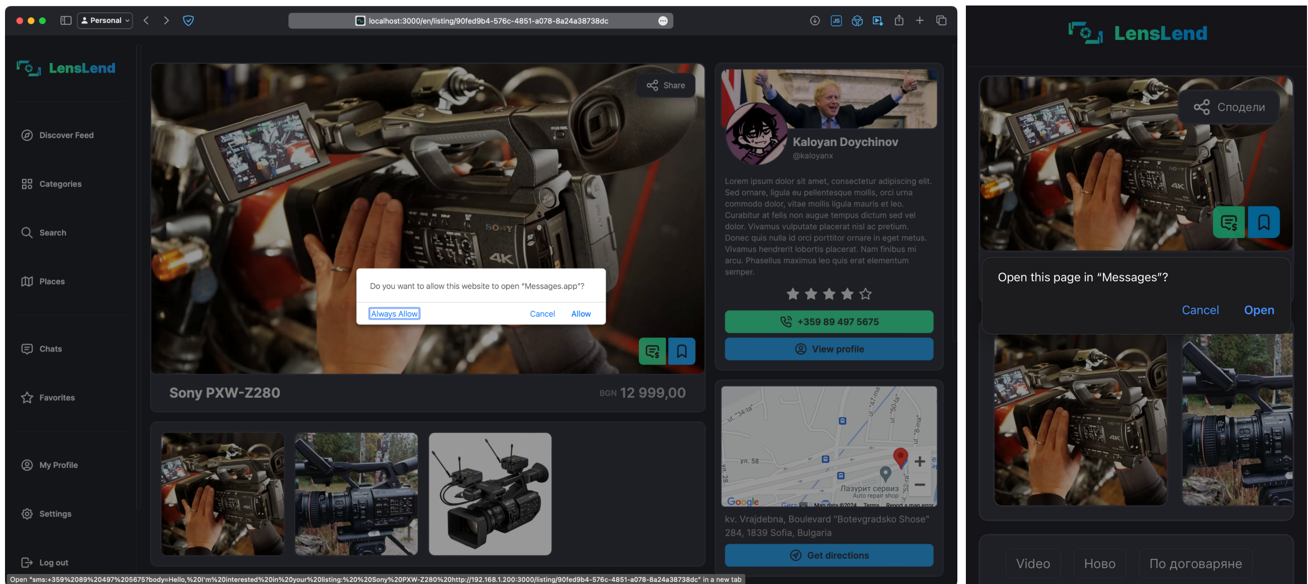
Всяка обява има 2 важни бутона за извършване на действие – запазване на пост и изпращане на съобщение. При запазването на даден пост, се отваря модал – изскачащ прозорец.



фиг. 4.36. – Изскачащ прозорец след запазване на публикация

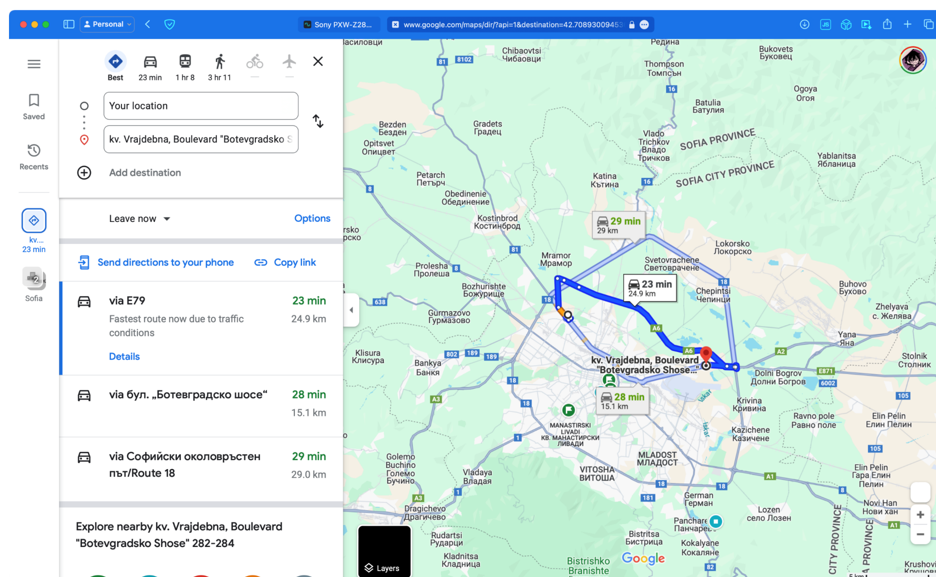
Другият бутон е за изпращане съобщение до потребителя. Линкът е форматиран за изпращане на дадено SMS съобщение. Като след натискане в повечето модерни браузъри излиза изскачащ прозорец,

който пита потребителя за потвърждение, дали иска да бъде отворено тяхното приложение за изпращане на съобщения по подразбиране.



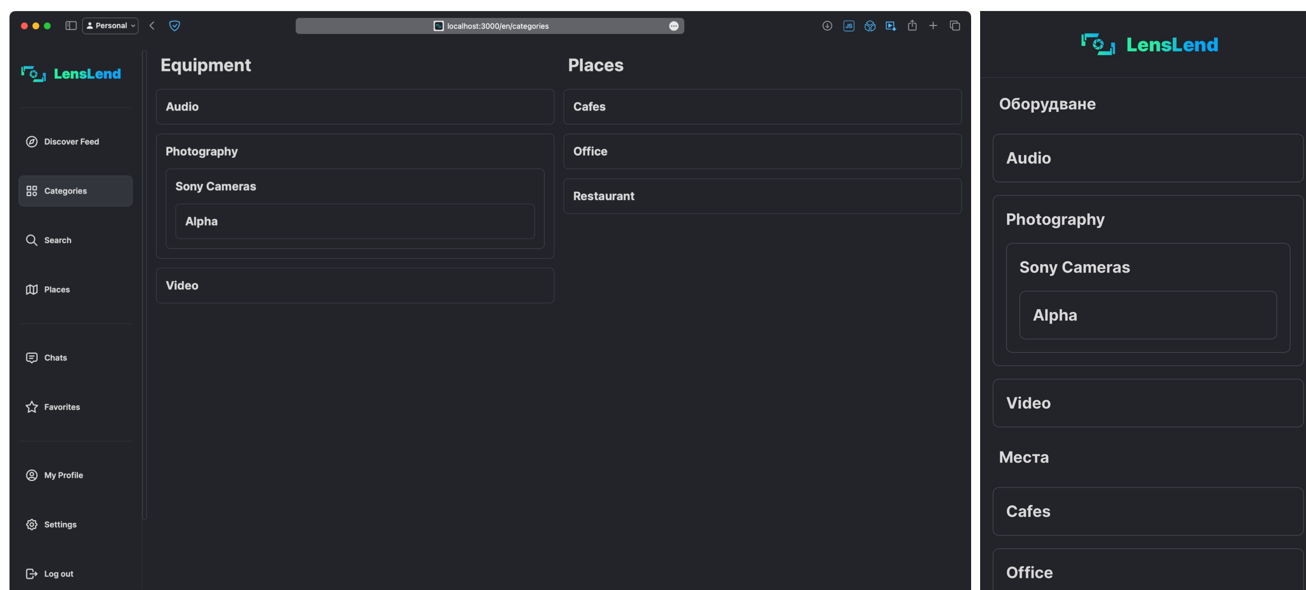
фиг. 4.37. – Изскачащ прозорец на браузъра след натискане на бутона за изпращане на съобщение

Подобен на него бутон е този за повикване, който се намира в кутийката на потребителя. На картата, показана под потребителя също има и бутон за получаване на навигация в Google Maps. След неговото натискане се отваря на нова страница (или в приложението, ако потребителят е с тел.) – Google Maps с готов маршрут от текуща локация.



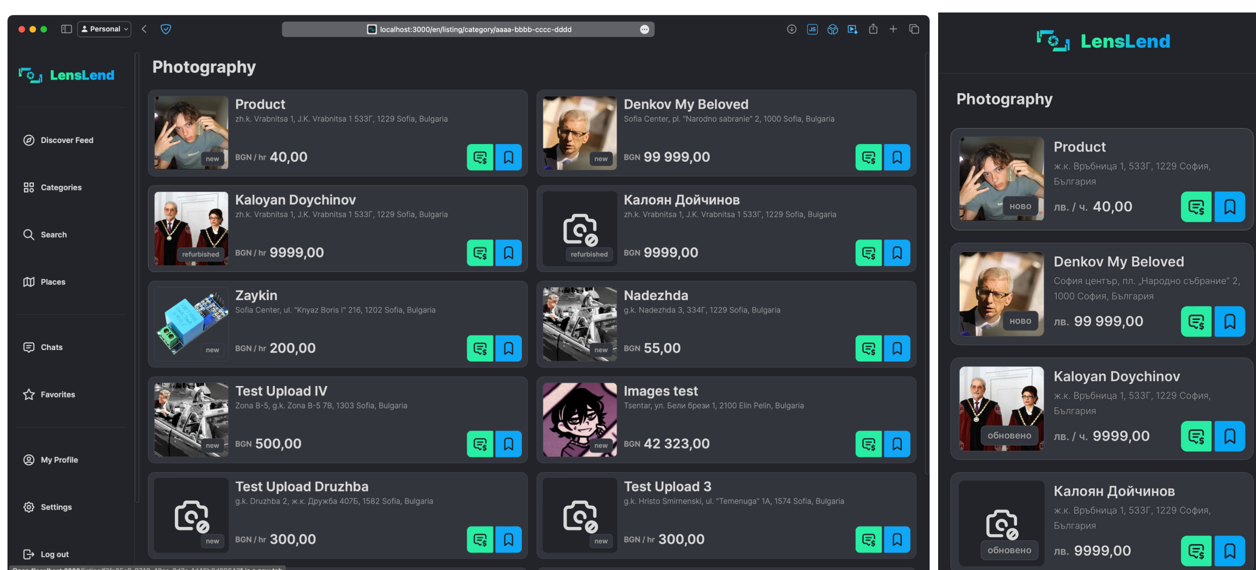
фиг. 4.38. – Google Maps навигация

Втората страница, която се намира в навигационното поле е тази за избор на категории и подкатегории. В нея са изброени всички категории и за обяви, и за места.



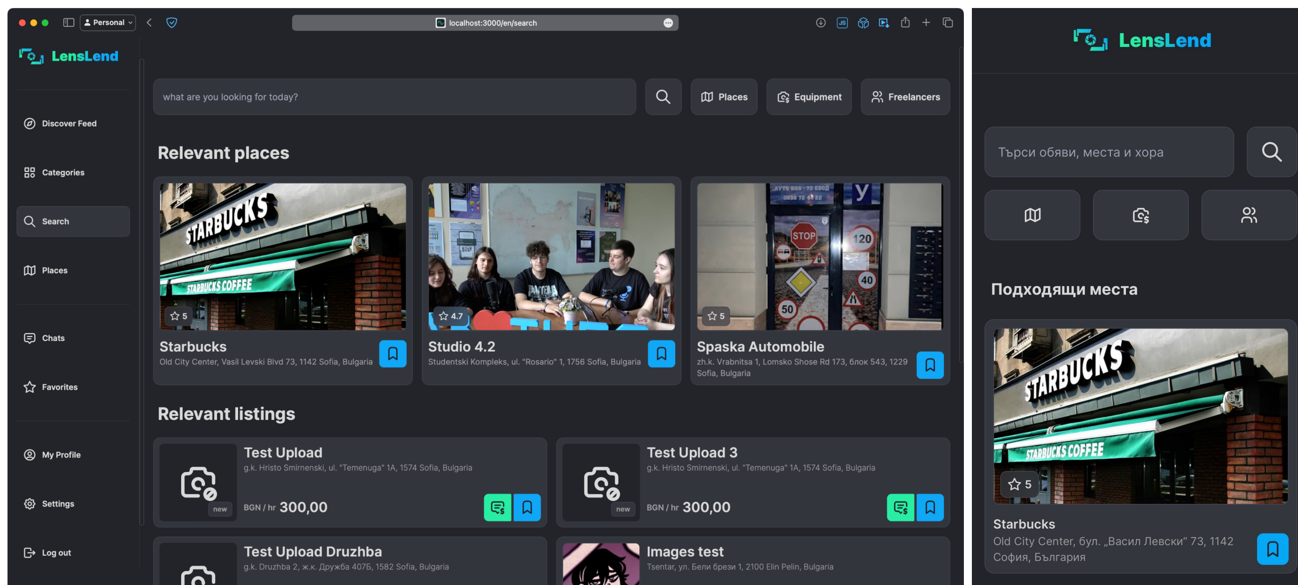
фиг. 4.39. – Екран със списък от категории

При избиране на категория се отива на нова страница, в която са изброени обяви или места за дадената категория и нейните подкатегории. Тази страница също е разделена на страници и лимити (Pagination). Като контролите ѝ са като на *фиг. 4.33.*



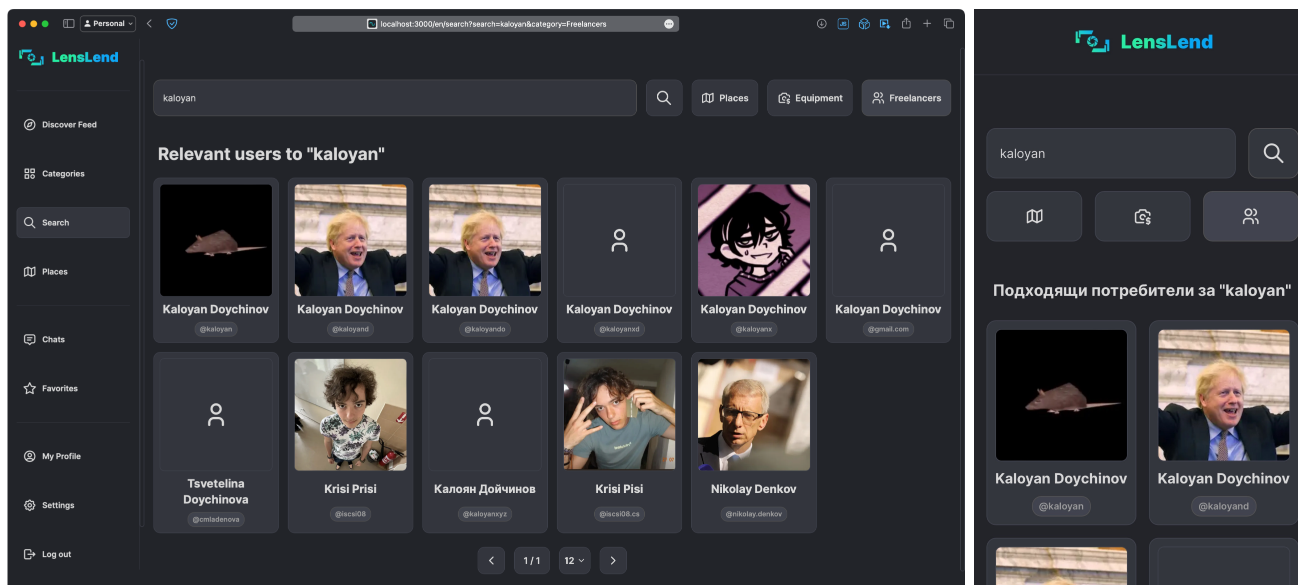
фиг. 4.40. – Страница с изброени елементи от дадена категория и нейни подкатегории.

Третата страница, спрямо навигационното поле, е търсачката на приложението. Тя може да се използва като глобална търсачка, като търси едновременно за всички елементи – места, обяви и потребители.



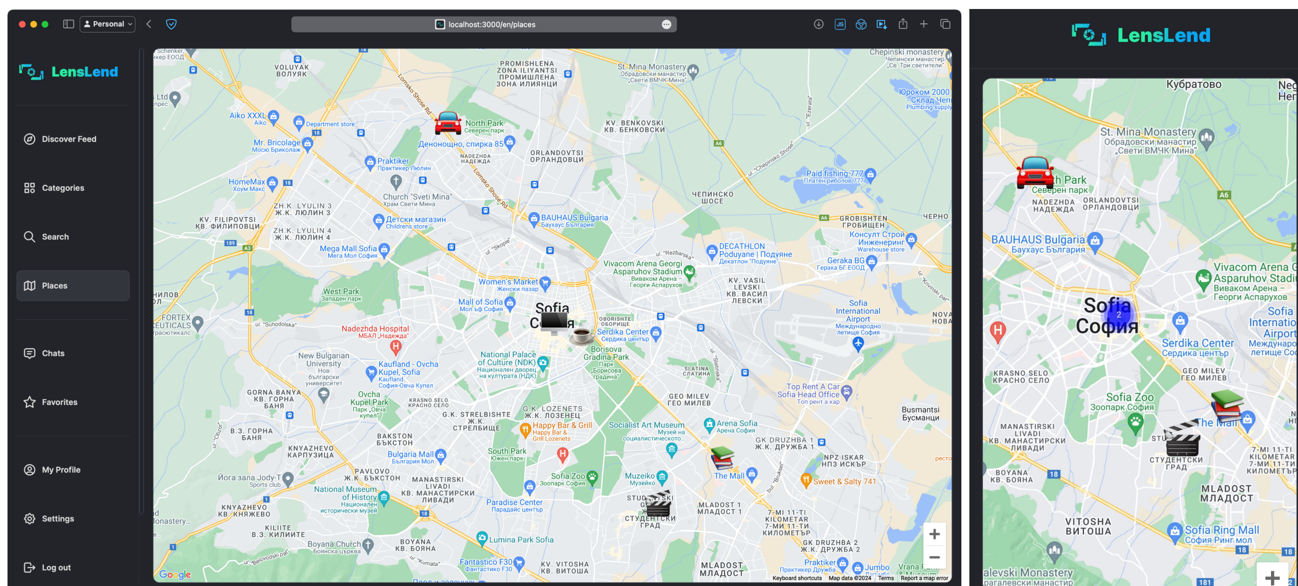
фиг. 4.41. – Изглед на търсачката, когато няма нищо потърсено или се използва като глобална търсачка

Бутоните до полето за търсене са за филтриране на елемент за търсене – място, обява или потребител. Интерфейсите за определено търсене са с много страници и лимити (Pagination).



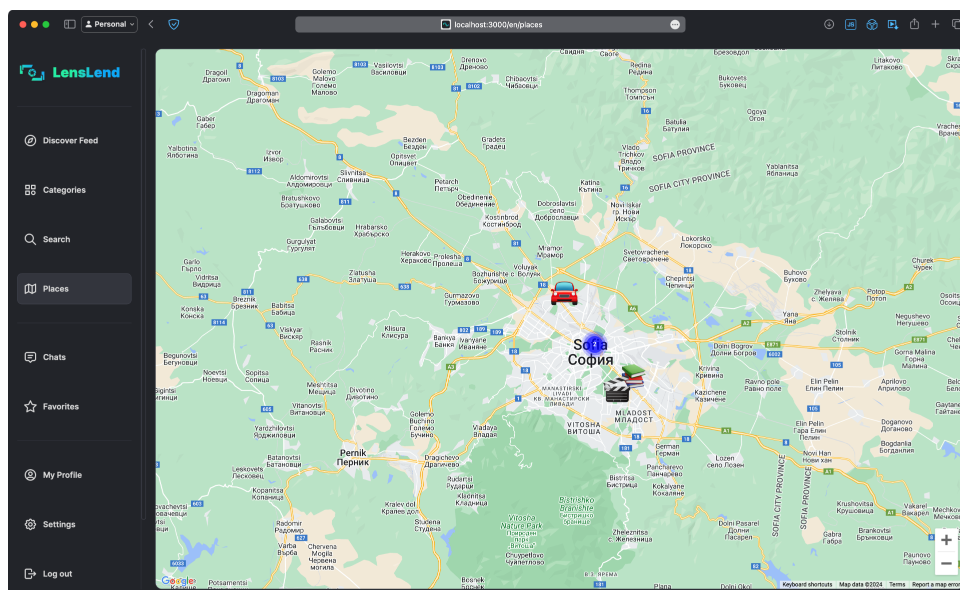
фиг. 4.42. – Търсене за определен елемент

Четвъртата страница от навигационното поле е тази за разглеждане и намиране на места. Идеята на страницата е да бъде визуална и затова тя е една голяма интерактивна карта, на която са показани всички места на дадено място с техните специални иконки.



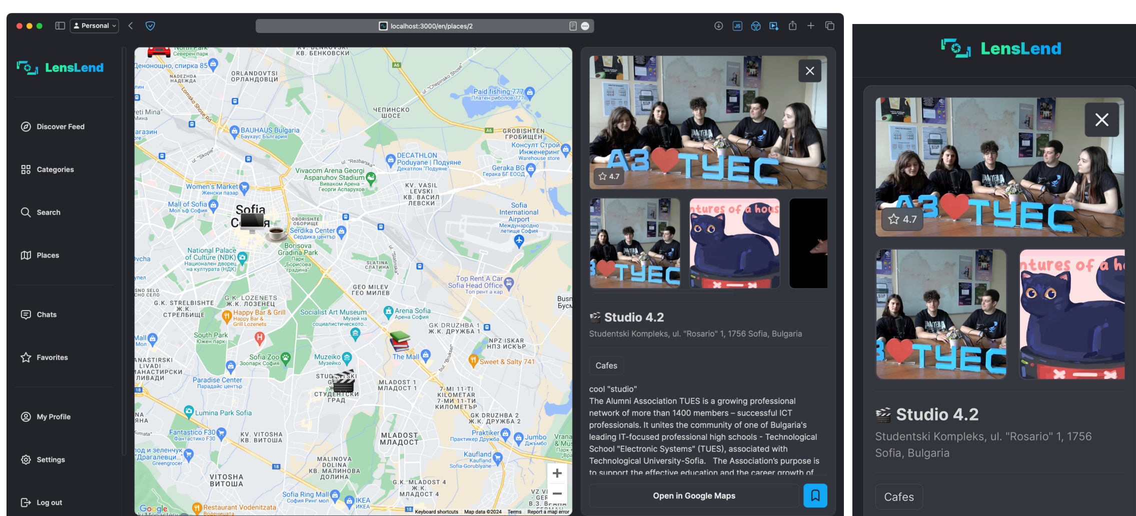
фиг. 4.43. – Разглеждане на карта с места

При събиране на повече места на дадено място, те се групират заедно в кръгче с число, което показва колко места са събрани на едно място, този метод се нарича “clustering”.



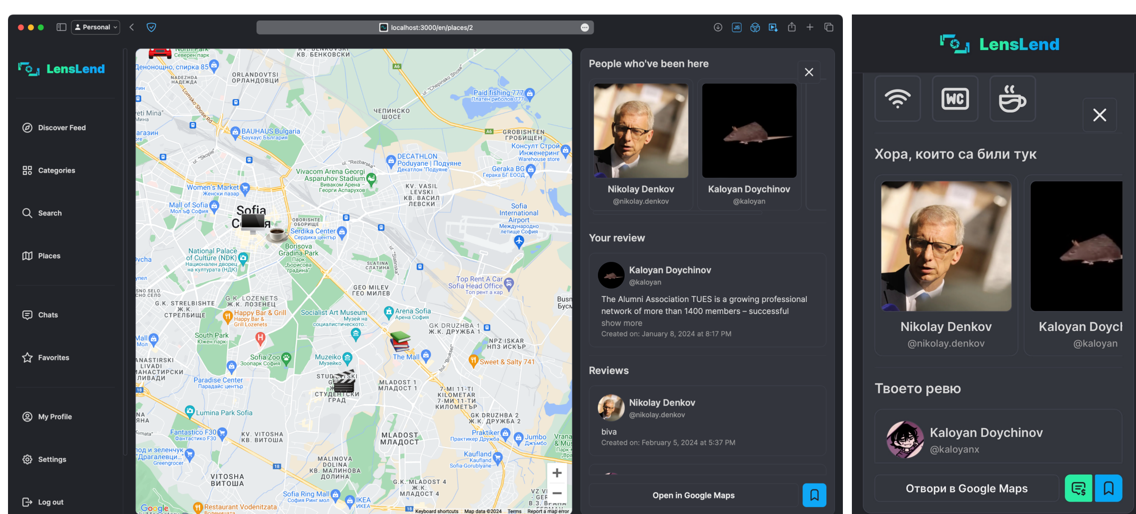
фиг. 4.44 – Отдалечен изглед на картата, с цел демонстриране на “clustering”

При натискане на някоя иконка на място, то се отваря встрани, като стеснява главната карта за разглеждане. А ако приложението се използва от телефон, то мястото бива насложено над самата карта. На частта за място се показва информация като: главна снимка, галерия от изображения, иконка, наименование, местоположение в текст, категория, описание, тагове, услуги, времеви статистики и много други, които се показват при плъзгане „scroll” надолу.



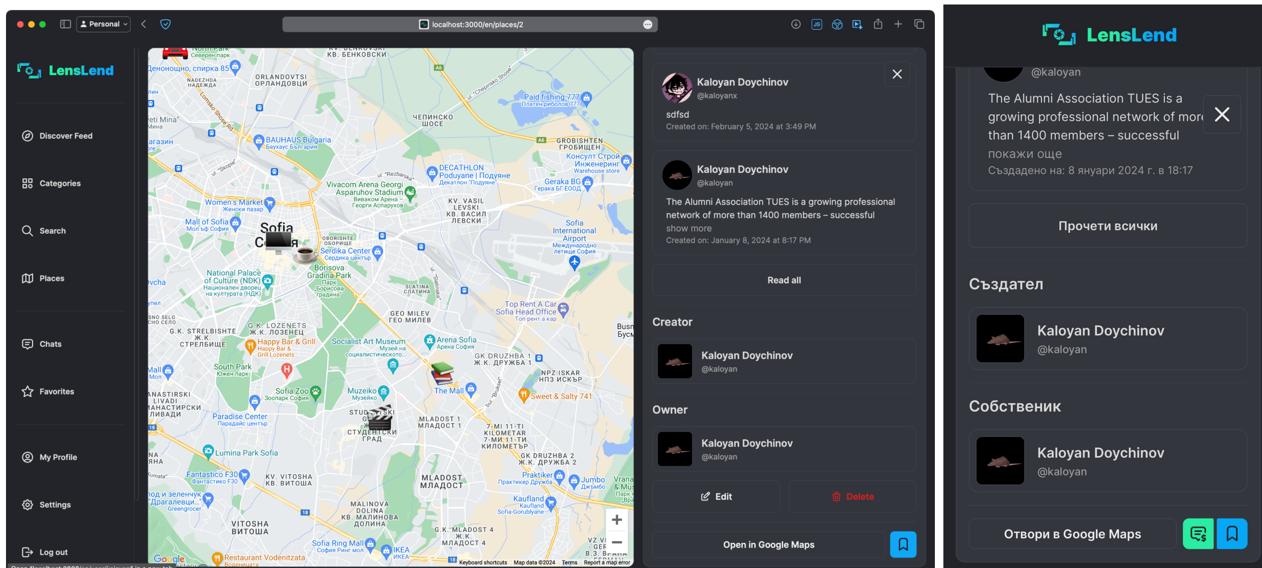
фиг. 4.45. – Първа част от изглед на място

Освен това са покани хората, посетили даденото място, топ три ревюта, както или ревюто на потребителя, ако той е оставил, или форма за поставяне на ревю и оценка.



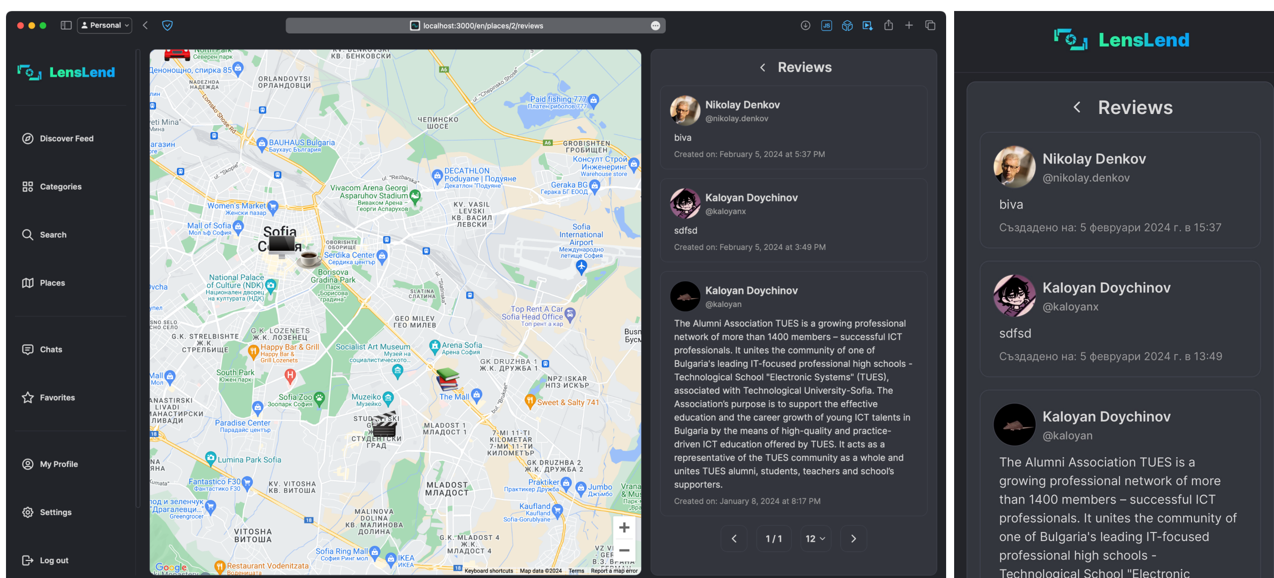
фиг. 4.46. – Втора част от изглед на място

Ако се плъзне “scroll” по-надолу може да се видят създателя и притежателя на мястото, ако има такъв. Както и ако потребителят е модератор, администратор или създател (притежател) може да види бутоните за извършване на модифициращи действия – изтриване или промяна на информацията, това също се отнася и за екрана на обяви.



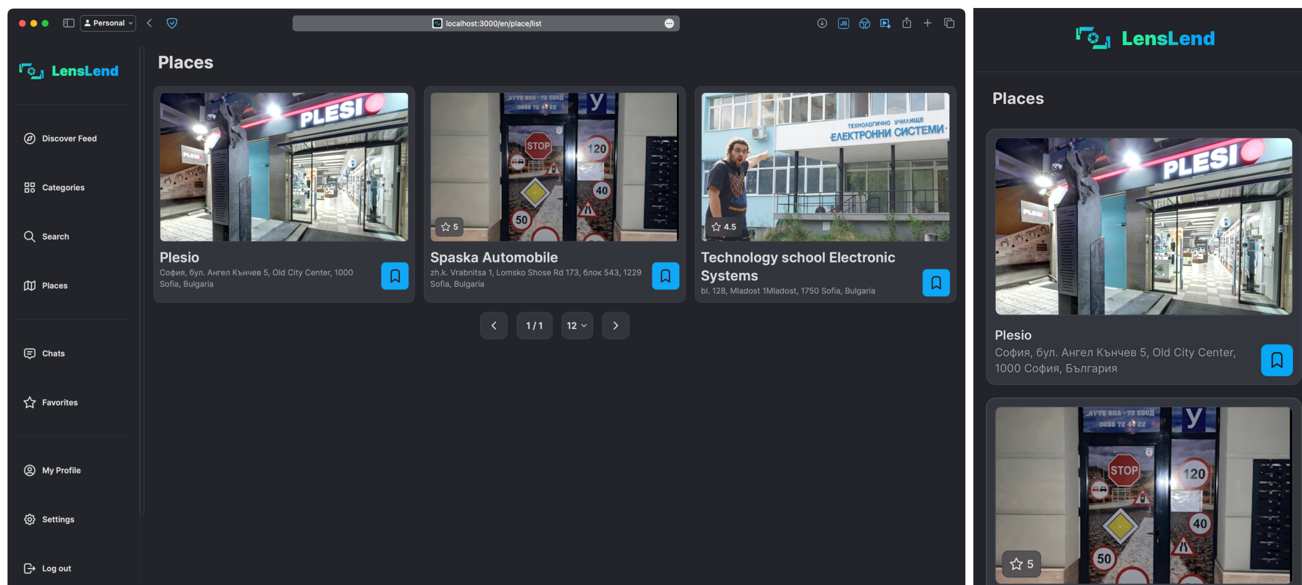
фиг. 4.47 – Трета част от изглед на място

При избиране на бутона за прочитане на всички ревюта, се преминава на нова страница, която също е встрани, за прочитане на всички ревюта по страници и с лимит (Pagination).



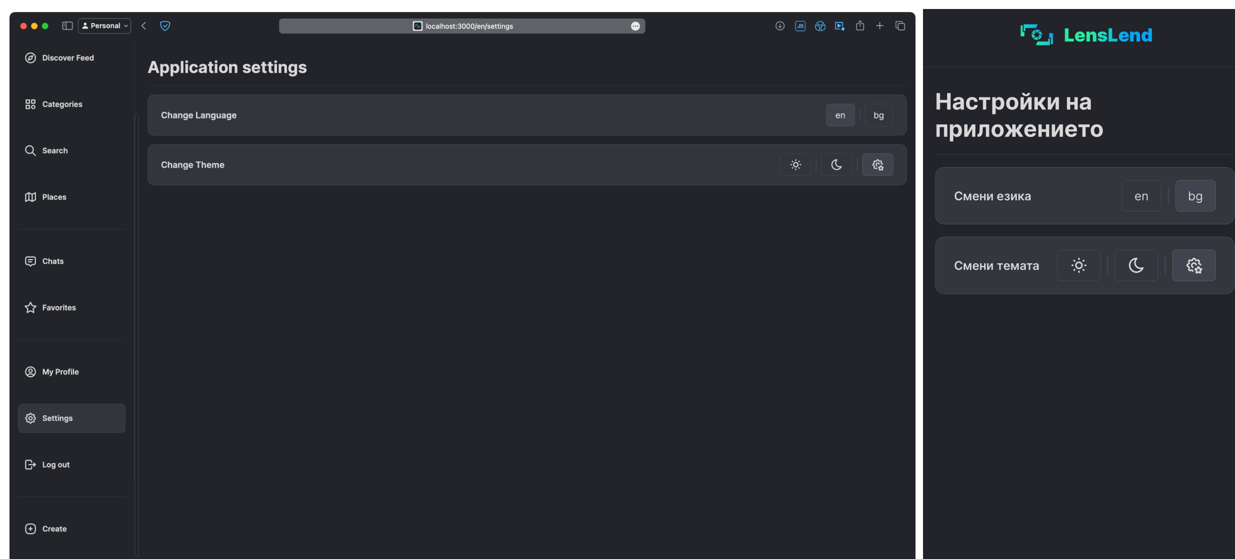
фиг. 4.48. – Изглед на ревюта

Освен интерактивната карта, съществува и екран за разглеждане на всички съществуващи места по страници с лимит (Pagination), като се натисне бутона за виждане на всички места в началната страница.



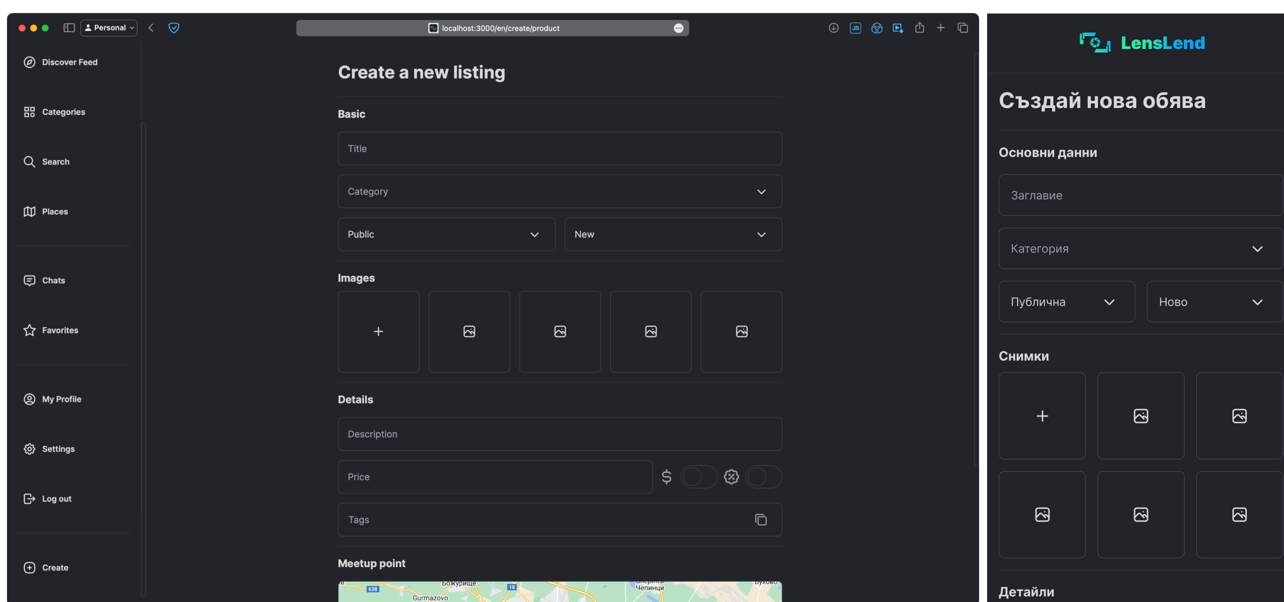
фиг. 4.49. – Страница за гледане на всички места под формата на картичка

Друга страница от навигационното поле е тази за настройките на приложението. В нея се съдържат две главни настройки – тази за избиране на език (български или английски), както и тази за избиране на тема (автоматична, светла или тъмна). Като тази информация се съхранява в локалното хранилище на уеб браузъра.



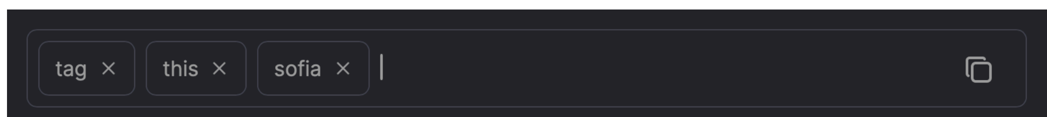
фиг. 4.50. – Екран за настройки на приложението

Страниците за създаване на съдържание (обяви или места) са защитени и не могат да биват достъпвани от нерегистрирани потребители. За създаване на обява има вход за заглавие, избиране на категория, видимост, състояние на продукта, избор на снимки (както и тяхната подредба с плъзгане (drag-n-drop). Също включва описание, цена, тагове (по подобие на YouTube), избор на локация, както и дали цената е за наем и дали може да се преговаря.



фиг. 4.51. – Страница за създаване на обява

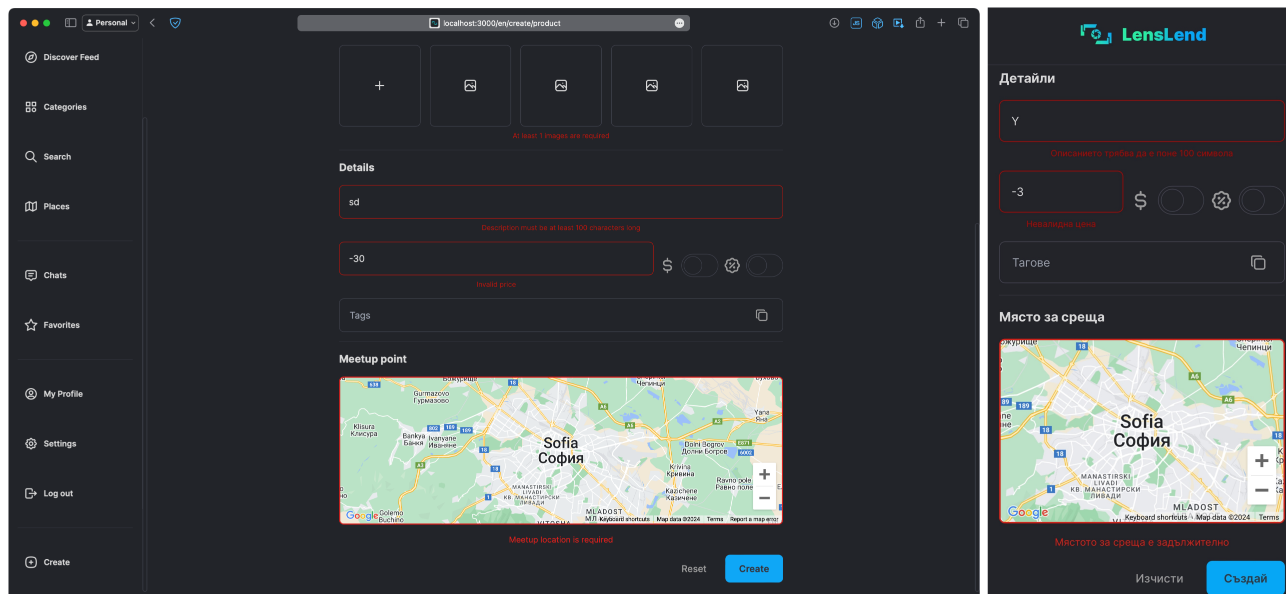
Таговете могат да се копират и поставят с бутона за копиране. Освен това те могат да бъдат изтривани както и тяхното „X" бутонче, така и с бутона за изтриване назад на клавиатурата „backspace".



фиг. 4.52. – Поле за добавяне на тагове за обява или място

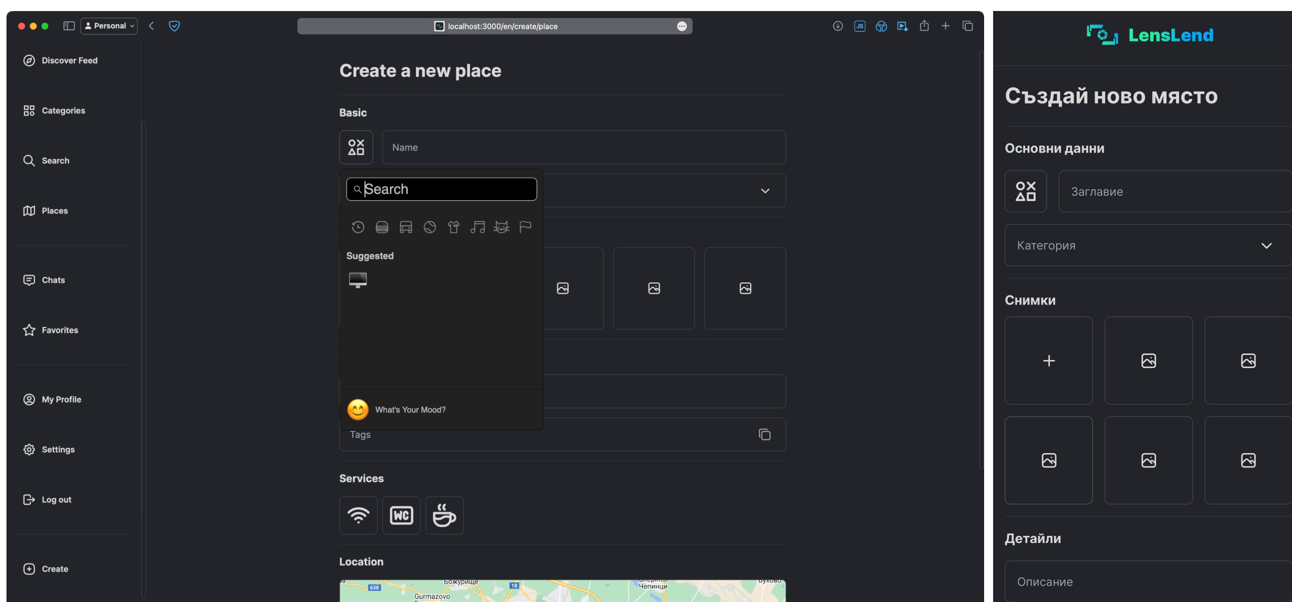
Освен това, както всички форми в приложението – за създаване на място, обява, коментар, ревю, както и регистрация или влизане, има

валидация и обратна връзка на полетата и формата за клиентски и сървърни грешки.



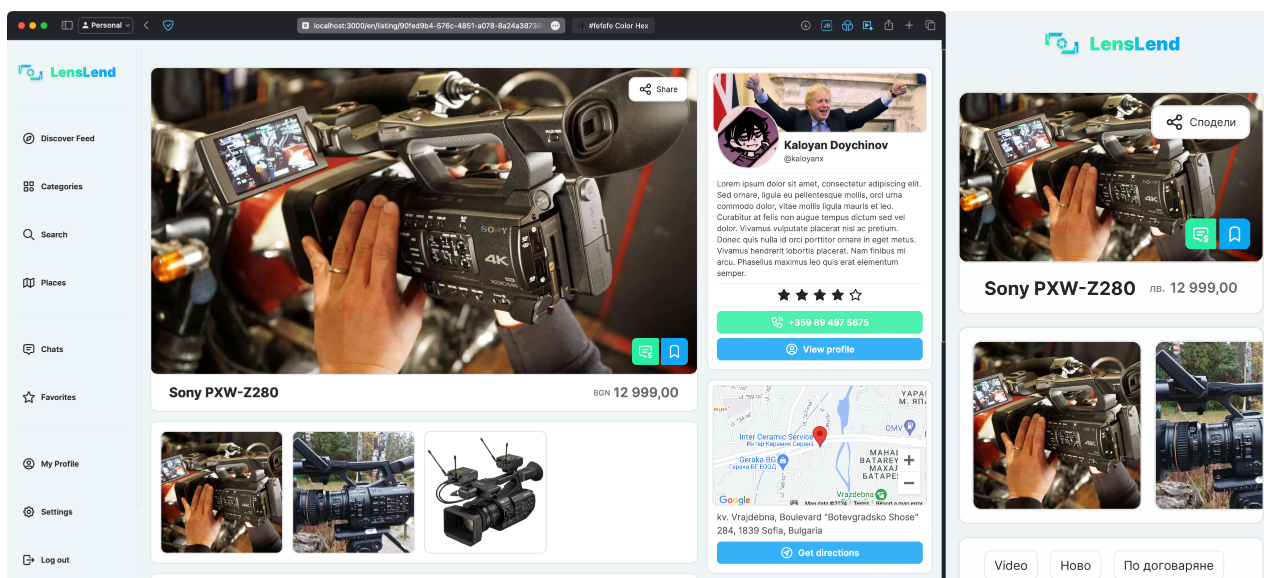
фиг. 4.53. – Грешки при подаване на грешни данни

Другата страница за създаване на съдържание е тази за създаване на място. В нея са изчезнали някои компоненти, но са се появили нови – за избиране на иконка на място (емоджи), както и за избиране на услуга от вече готов списък.



фиг. 4.54. – Страница за създаване на ново място с отворен прозорец за избиране на иконка на място

Освен промяна на езика (на български или на английски), приложението разполага и с опцията да се използва със светла или тъмна тема. Като също има функционалност и за автоматично сменяне на темите спрямо настройките на операционната система на потребителя.



фиг. 4.55. – Светла тема на потребителския интерфейс

Заключение

В изготвената дипломна работа всички заложени изисквания за функционалности са покрити и основният проблем беше решен. Освен основните изисквания, системата беше надградена и с много други. Целта им е да предоставят на потребителите още по-добро потребителско изживяване.

Въпреки пълно изпълнение на изискванията на тази дипломна работа, има още много работа напред за развиване на този проект. За бъдещо развитие на платформата са определени следните функционалности:

- Създаване на вътрешна чат система за комуникация между потребителите в платформата.
- Свързване на профила в платформата с OAuth - Google, Apple, Facebook и други популярни платформи.
- Интегриране на опции за плащане по подобие на eBay - онлайн плащане чрез Stripe или наложен платеж
- Създаване на администраторски панел, в който да се създават нови категории и услуги. Да има възможност за промяна и изтриване на списък от потребители, места или обяви.
- Създаване на функционалност за промяна на информация за места от обикновени потребители (не администратори или модератори) с исторически преглед по подобие на Уикипедия.
- Създаване на функционалност за подаване на сигнал за обява, място или потребител.
- Интегриране на DeepL в потребителския интерфейс или подобна технология за превеждане на заглавия и описания, спрямо езика на потребителя.

Използвана литература

- https://docs.nestjs.com/
- https://nextjs.org/docs
- https://www.prisma.io/docs/orm
- https://developer.mozilla.org/
- https://react.dev/reference/react
- https://www.postgresql.org/docs
- https://next-intl-docs.vercel.app/docs
- https://visgl.github.io/react-google-maps/docs
- https://swr.vercel.app/docs
- https://axios-http.com/docs/intro
- https://zod.dev
- https://redis.io/docs/
- https://docs.aws.amazon.com/s3/
- https://developers.google.com/maps/documentation
- https://tailwindcss.com/docs/installation
- https://fajarwz.com/blog/web-rendering-what-is-ssr-csr-ssg-and-isr/
- https://aalonso.dev/blog/how-to-generate-generics-dtos-with-nestjsswagger-422g
- https://github.com/vercel/next.js/tree/canary/examples
- https://github.com/nextauthjs/next-auth-example
- https://xerosource.com/nextjs-authentication-with-external-api/
- https://princekfrancis.medium.com/aws-s3-serverless-image-upload-and-thumbnail-creation-6b4fc1ed98b3
- https://www.youtube.com/watch?v=bGzanfKVFeU - React конференция
- https://stackoverflow.com/questions/71917408/how-to-check-for-roles-in-nestjs-guard

- https://youtu.be/MTcPrTIBkpA?si=AQfXCo2NSn1DOfDZ - различни типове генериране и изобразяване на страници (NextJS)
- https://tailwindcss.com/blog/automatic-class-sorting-with-prettier
- https://apetools.webprofusion.com/
- https://github.com/hacktues-9/ht-client
- https://github.com/hacktues-9/tf-client
- https://github.com/KokosTech/snackoverflow
- https://github.com/KokosTech/imgurish
- https://github.com/KokosTech/imgurish-frontend
- https://github.com/KokosTech/graphics

---

[i] OLX – https://olx.bg, други версии на платформата също могат да бъдат открити на следните адреси: https://olx.ro, https://olx.pl, https://olx.ua, https://olx.pt

[ii] Базар.БГ - https://bazar.bg

[iii] В сравнения с платформи като OLX.bg или последните дизайнерски тенденции в разработката на уеб приложение. Това не намалява по никакъв начин производителността и простотата в използването на приложението.

[iv] Проучване за предпочитание на платформа за онлайн продажби на стоки втора ръка от анонимни потребители на социалната мрежи Reddit - http://tinyurl.com/olxbg-bazarbg-1, http://tinyurl.com/olxbg-bazarbg-2

[v] Метаинформация (от англ. – meta information) – допълнителна информация, която пояснява даден обект (напр. заглавие, описание, тагове и др. на обява)

[vi] HTTP(S) – Hyper Text Transfer Protocol (Secure) – протокол, използван за предаване на хипертекстови документи, които позволяват изграждането на взаимосвързани информационни системи в интернет

[vii] WebSocket - протокол, който осигурява двупосочна, постоянна връзка между клиент и сървър по интернет, позволявайки обмен на данни в реално време. Често използван за чат системи в интернет, следене на акции, табла с времена и други.

[viii] gRPC – рамка за междусървърна комуникация, разработена от Google, която улеснява бързия обмен на данни между клиенти и сървъри използвайки HTTP/2 и протоколни буфери.

[ix] SEO – Search Engine Optimization – процес на оптимизиране на уебсайтове за подобряване на тяхната видимост в резултатите на търсачките (Google, Bing, DuckDuckGo, Brave и други), с цел привличане на повече органичен (неплатен) трафик.

[x] Статичен уебсайт – Static Website или Statically Generated Website (Статично генериран уебсайт)

[xi] Сървърен уебсайт – Server-Side Website или Server Rendered Website (Сървърно генериран или изобразен уебсайт)

[xii] Сървърно-изобразени – server-rendered

[xiii] Уеб приложение с една страница – Single Page Application или по-добре познато като SPA

[xiv] SSR – Server-Side Rendering – сървърно генерирано (изобразено) съдържание / компонент

[xv] CSR – Client-Side Rendering – клиентско генерирано (изобразено) съдържание / компонент

[xvi] API-Gateway – (ППИ шлюз) управляващ слой, който действа като единна точка на вход за всички клиентски заявки към микроуслуги в заден план, осигурявайки маршрутизация, агрегация на данни и защита на интерфейсите.

[xvii] HTML – https://developer.mozilla.org/en-US/docs/Web/HTML

[xviii] Маркиращ език – markdown - система от анотации на документ, която е синтактично различна от текста и указва какво да бъде оформлението на страницата

[xix] CSS – https://developer.mozilla.org/en-US/docs/Web/CSS

[xx] JavaScript (ECMAScript) – https://developer.mozilla.org/en-US/docs/Web/JavaScript#

[xxi] TypeScript – https://www.typescriptlang.org

[xxii] ReactJS – https://react.dev

[xxiii] Vue.js – https://vuejs.org

[xxiv] NextJS – https://nextjs.org

[xxv] Python – https://www.python.org

[xxvi] Django – https://www.djangoproject.com

[xxvii] ExpressJS – https://expressjs.com

[xxviii] NestJS – https://nestjs.com

[xxix] MySQL - https://www.mysql.com

[xxx] PostgreSQL – https://www.postgresql.org

[xxxi] Visual Studio Code – https://code.visualstudio.com

[xxxii] JetBrains WebStorm – https://www.jetbrains.com/webstorm/

[xxxiii] Рефакторинг - процес на модифициране и преструктуриране на код на даден софтуер, за да се подобри неговата структура, производителност и четимост, без да се променя неговата функционалност.

[xxxiv] Zed - https://zed.dev

[xxxv] Dependency Injection – инжектиране на зависимости - дизайн модел, който позволява на класовете да получават своите зависимости отвън, вместо да ги създават самостоятелно.

[xxxvi] Анотирани - @ - употребата на декоратори за маркиране на класове, методи и свойства с метаданни.

[xxxvii] Prisma ORM – https://www.prisma.io/orm

[xxxviii] ORM – Object-Relational Mapping - процес за свързване и управление на данни между обектно-ориентирани програмни езици и релационни бази данни.

[xxxix] DX – Developer Experience – подобно на потребителското изживяване, но в контекст на разработването на даден проект – от страна на разработчика.

[xl] Full Text Search – техника за търсене в база данни, която позволява намирането на записи, съдържащи текст, базиран на съвпадение с даден термин или фраза, вместо на точно съответствие.

[xli] DOM – Document Object Model – е програмен интерфейс за HTML и XML документи, който представя документа като структура от обекти (дърво).

[xlii] Меморизация – техника за оптимизация на производителността, която запазва резултатите от тежки функции и ги използва отново при повтарящи се входни параметри. Така се избягват повтарящи се тежки изчисления.

[xliii] Ревалидация – процес, който позволява на приложенията да поддържат своите статично генерирани страници актуални, като се използва техниката на инкрементално статично регенериране (ISR). Т.е. извличане на данни през определен период от време или при случването на дадено събитие – тригер на таг (с NextJS 13+).

[xliv] Контекст – React Context – функционалност, предоставена от ReactJS, която позволява на данните да бъдат лесно споделяни между компонентите на уеб приложение без необходимостта да се подават свойства (props) на дълбоки нива през всеки компонент от йерархия.

[xlv] Next-intl – https://next-intl-docs.vercel.app

[xlvi] @vis.gl/react-google-maps https://github.com/visgl/react-google-maps

[xlvii] google-map-react - https://www.npmjs.com/package/google-map-react

[xlviii] SWR – https://swr.vercel.app

[xlix] Axios – https://axios-http.com

[l] JSON – https://www.json.org/json-en.html

[li] Zod – https://zod.dev

[lii] JWT – https://jwt.io

[liii] Redis – https://redis.io

[liv] Amazon AWS S3 – https://aws.amazon.com/s3/

[lv] CDN – мрежа от разпределени сървъри, която доставя уеб съдържание и други уеб ресурси

[lvi] Google Maps Platform – https://mapsplatform.google.com

[lvii] TailwindCSS – https://tailwindcss.com

[lviii] Tabler Icons – https://tablericons.com

[lix] Prettier – https://prettier.io

[lx] ESLint – https://eslint.org

[lxi] Postman – https://www.postman.com

[lxii] Figma – https://www.figma.com

[lxiii] Git – https://git-scm.com

[lxiv] система за контрол на версиите – VCS – Version Control System – софтуер, който управлява различни версии на документи, програмен код и други информационни обекти, позволявайки за следенето на историята на промените.

[lxv] GitHub – https://github.com

[lxvi] WakaTime – https://wakatime.com

[lxvii] Фрийланс (от англ. – freelance) – работа на свободна практика - https://bg.wikipedia.org/wiki/Фрийлансър

[lxviii] Generic – Generic в TypeScript позволява дефинирането на компоненти, които могат да работят с различни типове данни, увеличавайки тяхната повторна употреба и гъвкавост.

[lxix] CRUD – означава буквално: създаване (Create), четене (Read), обновяване (Update) и изтриване (Delete) - основни операции за работа с данни в бази данни.

[lxx] Paginated – разделено на страници, метод за организиране на големи количества данни в удобни за преглед страници.